Received: 25 August 2021 | Revised: 21 October 2021 | Accepted: 14 November 2021

DOI: 10.1002/spe.3058

RESEARCH ARTICLE

WILEY

AuctionWhisk: Using an auction-inspired approach for function placement in serverless fog platforms

David Bermbach[1] | Jonathan Bader[2] | Jonathan Hasenburg[1] | Tobias Pfandzelter[1] | Lauritz Thamsen[2]

[1]Mobile Cloud Computing Research Group, TU Berlin & Einstein Center Digital Future, Berlin, Germany

[2]Distributed and Operating Systems Research Group, TU Berlin, Berlin, Germany

Correspondence
David Bermbach, TU Berlin, Sekr. EN17, Einsteinufer 17, D-10587 Berlin, Germany.
Email: db@mcc.tu-berlin.de

Abstract
The Function-as-a-Service (FaaS) paradigm has a lot of potential as a computing model for fog environments comprising both cloud and edge nodes, as compute requests can be scheduled across the entire fog continuum in a fine-grained manner. When the request rate exceeds capacity limits at the resource-constrained edge, some functions need to be offloaded toward the cloud. In this article, we present an auction-inspired approach in which application developers bid on resources while fog nodes decide locally which functions to execute and which to offload in order to maximize revenue. Unlike many current approaches to function placement in the fog, our approach can work in an online and decentralized manner. We also present our proof-of-concept prototype AuctionWhisk that illustrates how such an approach can be implemented in a real FaaS platform. Through a number of simulation runs and system experiments, we show that revenue for overloaded nodes can be maximized without dropping function requests.

KEYWORDS
distributed scheduling, fog computing, function-as-a-service, serverless computing

1 | INTRODUCTION

Recently, the paradigm of fog computing has received more and more attention. In fog computing, cloud resources are combined with resources at the edge, that is, near the end user or close to IoT devices, and in some cases also with additional resources in the network between cloud and edge.[1] While this adds complexity for applications, it comes with three key benefits: First, leveraging compute resources at or near the edge can lower response times, which is crucial for application domains such as autonomous driving or 5G mobile networks.[1,2] Second, data can be filtered and preprocessed early on the path from edge to the cloud, which reduces the data volume.[3] Especially in IoT use cases, it is often not feasible to transmit all data to the cloud as the sheer volume of produced data exceeds the bandwidth capabilities of the network[4] or leads to significant energy consumption for wide-area networking.[5] Third, keeping parts of applications and data at the edge can help to improve privacy, for example, by avoiding “centralized data lakes”[6] in the cloud. Overall, fog computing, thus, combines the benefits of both cloud and edge computing.

While there are many open research questions in fog computing, a key question has not been answered yet: Which compute paradigms will future fog applications follow? In previous work,[1] we argued that a serverless approach—which we understand as Function-as-a-Service (FaaS) in this article—is a good fit for the edge. The main reason for this is that resources at the edge are regularly considerably constrained so that provisioning them in small function slices is more efficient than provisioning them using virtual machines or long-running containers. Additionally, the idea of having strictly stateless functions, separated from data management,[7,8] supports moving parts of applications seamlessly between edge and cloud resources.

Now, assuming a serverless world in which application components can run as functions on FaaS platforms in the cloud, at the edge, and medium-sized data centers in between, the question of how to distribute fog application components can be reduced to the issue of function placement across multiple geo-distributed sites. In Reference 9, we introduced the idea of using auction-inspired mechanisms for function placement and evaluated it in small-scale simulation experiments. We were able to confirm our assumptions that this would be an efficient approach to decentralized scheduling of functions over a distributed fog infrastructure and have argued for further work in this area. In this article, we extend our previous work and make the following contributions:

1. We describe a more general conceptual approach of using a decentralized auction scheme to control function placement (Section 3).
2. We discuss practical engineering challenges for implementing this approach in existing FaaS platforms (Section 4)
3. We present a simulation tool as well as the underlying system model and use it to study the effects of various parameters on our auction-based function placement (Section 5).
4. We implement our approach as a proof-of-concept prototype called AuctionWhisk based on Apache OpenWhisk and evaluate our prototype through experiments (Section 6).
5. We critically discuss the limitations of our work and identify future research directions in the field (Section 7).

2 | BACKGROUND

In this section, we introduce and describe fundamental concepts of FaaS, fog computing, and auctions. As all three topics have received major attention in research in the last few years and different definitions have emerged, we want to clarify the terminology we adopt in this article. We also give a short overview of Apache OpenWhisk which we used as basis for our prototype system.

2.1 | Function-as-a-Service (FaaS)

Within the field of cloud computing, the Function-as-a-Service (FaaS) paradigm has emerged as the latest evolution in resource sharing. The FaaS programming model facilitates highly scalable event-driven applications.

Developers deploy their code in the form of functions to a FaaS platform that handles code invocation and scaling, lowering the management burden for the consumer. Here, a function is a piece of business logic that is executed in response to an event. Functions can be implemented in any programming language as long as a runtime environment for the language is supported by the target FaaS platform. Events can be web requests, monitoring data, or even IoT sensor readings, thus making the FaaS approach a versatile option for many use cases. Logically, these functions live only as long as they process a single event and are reset with every invocation. This is usually achieved with lightweight virtualization techniques such as containerization, microVMs, or unikernels,[10-12] which enable FaaS platforms to spin up and destroy isolated instances quickly. As a result, no state can be persisted within a function across multiple invocations, they are thus referred to as “stateless”. This is also one of the main reasons for their scalability: A FaaS platform can spin up multiple concurrent instances of a function to handle concurrent events, while quickly shutting them down to free up resources for other functions. The same characteristic makes FaaS functions a good fit to fog and edge environments:[1] When migrating functions, there is no need to handle session state within the function container. Instead, the function container can simply be terminated on the original node and restarted on the new node. In practice, FaaS functions are usually used in conjunction with other platform services, especially in combination with storage and database services that manage state, for example Reference 13.

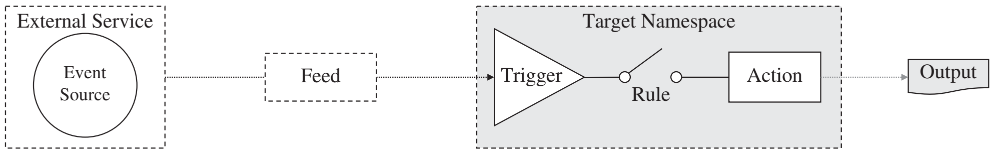


FIGURE 1 The OpenWhisk programming model[18]

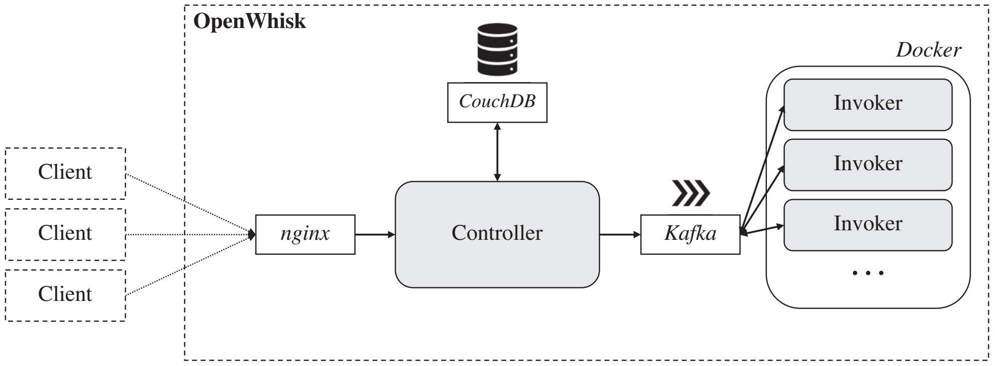


FIGURE 2 Components of an OpenWhisk deployment[18]

While scalability and low management overhead are clear advantages for developers, operators have the benefit that their infrastructure can be leveraged more efficiently. Instead of allocating coarsely grained resources in the form of virtual machines or containers statically, these more finely grained functions can be more efficiently mapped to underlying infrastructure and moved dynamically.[14-16] The platform provider can then charge tenants based on actual usage. In 2021, all major cloud service providers offer such a FaaS platform, for example, AWS Lambda[*], Microsoft Azure Functions[†], Google Cloud Functions[‡], or IBM Cloud Functions[§].

2.2 | Apache OpenWhisk

In addition to cloud-hosted FaaS platforms, a number of options for FaaS platforms have also emerged in research and open source communities, for example, tinyFaaS,[10] OpenLambda,[15] or SAND.[17] Another noteworthy example is Apache OpenWhisk, which is also at the core of IBM's Cloud Function service.[7]

The OpenWhisk platform allows for custom functions, called *actions*, to be executed in response to an event, the *trigger*. Figure 1 gives a high-level overview of the programming model as shown in the official OpenWhisk documentation.

Similarly, Figure 2 gives an overview of the OpenWhisk components: At the core of the OpenWhisk platform, the *Controller* provides endpoints to process triggers and schedules action invocations. The *Invokers* use Docker containers to isolate different actions, with a new container being spawned for each execution request. A *Kafka* queue serves as a buffer between Controller and Invoker, for example, to avoid unnecessary cold starts.[19] Additionally, a *CouchDB* instance holds actions, triggers, rules, and user-related information such as credentials or namespaces; *nginx* is used as the HTTP endpoint for the platform.

A complete installation of the OpenWhisk platform requires a substantial amount of resources which may not always be available to the FaaS platform, especially when moving toward the edge. For this reason, there is also a more lightweight version of OpenWhisk—*Lean* OpenWhisk. Lean OpenWhisk removes components such as the CouchDB database and Kafka queue in order to leave more resources for the function instances at the cost of some platform features.

[*]aws.amazon.com/lambda.
[†]azure.com/functions.
[‡]cloud.google.com/functions.
[§]www.ibm.com/cloud/functions.

2.3 | Fog computing

As the interest in fog computing increased, we have seen the emergence of different, often conflicting definitions of the term. In some cases, it is used as a synonym for edge computing, in others it is used to refer to resources that sit between edge and cloud or to all resources between cloud and device. In this article, we adopt the definition of Reference 1, where fog resources encompass all resources in edge, cloud, and in between, yet not those of end devices. We refer to these end devices also as “clients” and mean (embedded) IoT devices as well as mobile phones or computers, and to resources between edge and cloud as “intermediary nodes” or “intermediary” in short. We show an overview of cloud, edge, and fog computing in Figure 3.

This definition yields transparency from a client perspective, as clients simply access fog resources unaware of whether a particular service resides at the edge or in the cloud. Rather, they only see a singular “fog”.

In reality, the implementation of the fog is expected to follow a hierarchical model, where small edge nodes are collocated with access network equipment such as radio towers and fewer yet larger intermediary nodes are placed within the core network. Edge nodes will usually have the capabilities of single hosts such as Raspberry Pis or a small local cluster, while intermediaries are small- to medium-sized data centers. At the root of the tree-like model is the cloud with its seemingly infinite compute and storage resources. This leads to tradeoff decisions that have to be made for each application: on the one end, edge nodes can provide low latency, high bandwidth resource access for client devices, albeit with less available or more expensive resources. At the other end, the cloud provides scalable and (seemingly) infinite resources, but at the cost of large network distances to clients. Intermediary fog nodes fall somewhere in between.

2.4 | Fog-based FaaS platforms

We expect the paradigm of fog computing to enable entirely new classes of services and also to increase the quality-of-service (QoS)[20] of existing applications. In order to leverage the capabilities of the fog, it has been proposed to deploy applications on FaaS platforms that run on the fog infrastructure.[10,12,21-25]

Open source platforms such as OpenWhisk or OpenLambda[15] are a good fit for cloud nodes and larger intermediary nodes, yet their larger deployment overheads make them unsuitable for smaller intermediaries or the edge.[26] On more constrained nodes, Lean OpenWhisk and tinyFaaS[10] are possible options for function deployment, whereas NanoLambda is an option for on-device FaaS deployment.[22]

2.5 | Auctions

When demand for a product or service exceeds its available supply, auctions can be used to reach market equilibrium (i.e., to match suppliers with the optimal buyers). In an auction, all potential buyers enter bids which ideally correspond to the amount they are willing to pay. An auctioneer then sets auction rules which determine (i) how the winning bid(s) are chosen and (ii) which price the successful buyers have to pay.

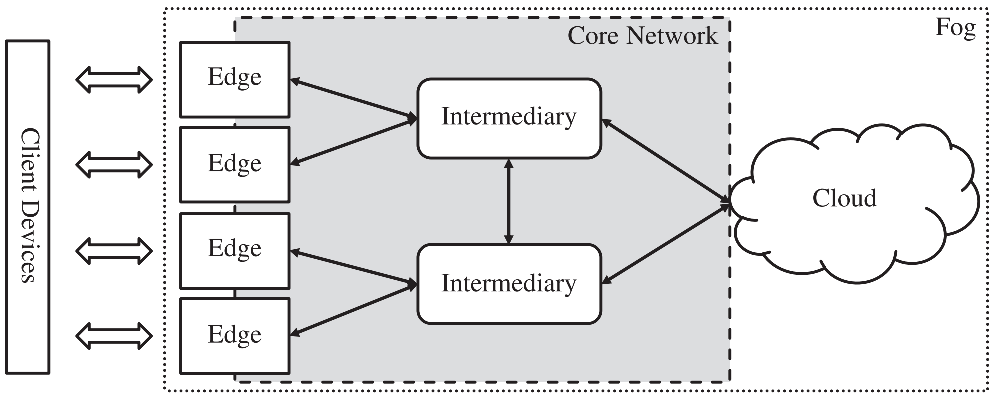


FIGURE 3 Overview of cloud, edge, and fog computing (adapted from Reference 1)

There is a large body of research on different auction rules. Here, we focus on the so-called sealed-bid auctions in which potential buyers enter their bids hidden from other potential buyers and the winner (or winners if there is more than one auctioned item) of the auction is the buyer with the highest bid. In these sealed-bid auctions, there are two fundamental rules for setting the price: in first-price auctions, the winner pays its bid; in second-price auctions, the winner pays the bid of the second highest (unsuccessful) potential buyer. The latter strategy is recommended whenever it is important that bids are truthful, that is, they correspond to the price that the potential buyers consider fair.

In the rest of this article, we will not expressly differentiate between different auction types but will implicitly assume either a first-price or second-price sealed auction. In the approach that we will present, it does not matter which pricing rule is used—they may affect prices but will not affect whether a placement decision is reached.

3 | AUCTION-BASED FUNCTION PLACEMENT

While a number of FaaS systems is already available, as discussed in Section 2.1, it is still not clear how to connect different FaaS deployments. Ideally, we would want edge, intermediary, and cloud deployments to coordinate function placement among each other. For instance, a request arriving in the cloud should probably still be processed in the cloud. A request arriving at the edge, however, should in most cases (exceptions include functions that require data located elsewhere[8]) be executed right at the edge. Only when the load on the edge node exceeds the available capacity should the request be delegated to an intermediary (and likewise from intermediaries toward the cloud). This concept, similar to cloud bursting, is based on the intuition of the cloud practically providing infinite resources.[27]

Based on this, we can conclude that function placement is straightforward when nodes have spare capacity (see also Reference 12 for a more general discussion) but becomes challenging when, for instance, an edge node is overloaded. In such a situation, the question is which request should be delegated to the next node on the path to the cloud as that request will incur extra latency. For reasons of resilience and fault-tolerance in a geo-distributed fog environment, such scheduling should ideally be managed in a decentralized manner, that is, through local decisions on each node. We can imagine a number of objectives and criteria for such local decisions, for example, to prefer short-running functions over long-running ones (and vice versa), to consider bandwidth impacts, to give some clients preference over others, or to prefer latency-critical functions over less critical ones.

In contrast to these, we propose to use an auction-inspired approach, which has been shown to lead to an efficient resource allocation in multiple domains (e.g., References 28-35): When application developers deploy their function to an integrated fog FaaS platform, they also attach two bids to the executable. The first bid is the price that the respective developer is willing to pay for a node to store the executable (in \$/s), the second bid is for the actual execution of the function (in \$/execution). In practice, both bids are likely to be vectors so that developers could indicate their willingness to pay more on edge nodes than in the cloud or even on a specific edge node. We explicitly distinguish bids for storage of the executable and the execution as edge nodes might encounter storage limits independently from processing limits. Both auctions closely resemble a sequence of first-price sealed-bid auctions as discussed in Section 2.5.

For ease of explanation, we use a first-price auction, although a second-price auction would be preferable in practice. Furthermore, as there is likely to be a minimum price equivalent to today's cloud prices, we can consider both bids a surcharge on top of a constant regular cloud price. For sake of clarity and since it has no impact on the bids, we leave this constant price aside in our explanation.

Storage bids: The nodes in our approach—edge, intermediaries, and cloud—analyze these bids and can make a local decision, that is, act as auctioneers, which is important for overall scalability and resiliency. We assume that cloud nodes will accept all bids that exceed some minimum base price. All other nodes will check whether they can store the executable. When there is enough remaining capacity, nodes simply store the executable together with the attached bids and start charging the application based on the storage bid. When there is not enough disk space left, nodes decide whether they want to reject the bid or remove another already stored executable. For this, nodes try to maximize their earnings. A simple strategy for this, comparable to standard bin packing, is to order all executables by their storage bid (either the absolute bid or the bid divided by the size of the executable) and remove stored executables until the new one fits in. Of course, arbitrarily comprehensive strategies can be used (in Section 5.5, we will show through simulation how changing this auctioning strategy affects results) and could even consider the processing bid and the expected number of executions. In the end, this leads to a situation where some or all nodes will store the executable along with its bids. See also Figure 4 for an overview of this auction step.

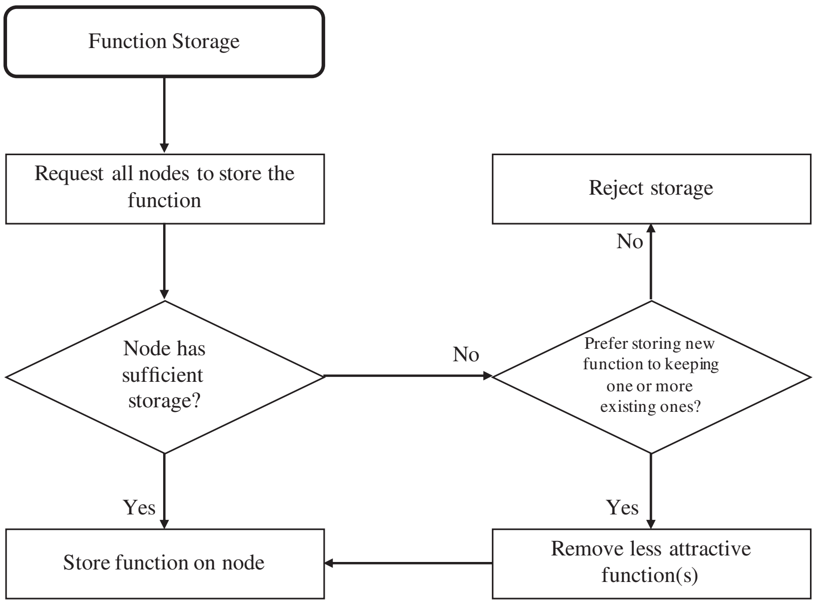

Function Storage
Request all nodes to store the function
Node has sufficient storage?
No
Yes
Store function on node
Prefer storing new function to keeping one or more existing ones?
No
Reject storage
Yes
Remove less attractive function(s)

FIGURE 4 Deployment and storage process of a single function

Please note that this process differs from standard auctions since it is not clear upfront how many items are auctioned off, since this depends on the size of the executables as well as the storage capacity of the respective node.

Processing bids: When a request arrives, nodes have to decide whether they *want* to process said request. A node *can* process a request if it stores the corresponding executable and if it has sufficient processing capacity to execute the function. We can imagine arbitrarily complex schemes for making the decision on whether to execute the request or not if an execution is possible. For instance, nodes might try to predict future requests (and decide to wait for a more lucrative one) or they might try to queue a request shortly—all with the goal of maximizing earnings. In fact, we believe that this opens interesting opportunities for future work.

For the explanations in this article, we will assume that a node will decide to process a request as soon as it is capable of doing so. This means that upon receipt of a set of incoming requests, a node will follow these four steps:

1. Reject all requests for which the executable is not stored locally.
2. Sort remaining requests by their bid (highest first) into a request list.
3. While capacity is left, schedule requests from request list.
4. Reject all requests that exceed the capacity.

Rejected requests are then pushed to the next node on the path to the cloud. As the cloud—by definition—stores all executables and has for all practical purposes unlimited capacity, all requests will at least be served in the cloud. See also Figure 5 for an overview of this auction step.

Please note that this auction step is different from standard auctions as well in that the number of auctioned items is unknown a-priori. Instead, it depends on the compute demands and execution duration of requested functions as well as available compute capacity on the respective node.

Overall, this approach solves the question of function allocation, all functions are executed as close as possible to the edge; under high load, it will also have the effect that prices increase toward the edge which is a good incentive considering the high CapEx of installing an edge infrastructure. Of course, the effects of this allocation scheme also depend on the behavior and strategies of tenants and nodes in the system, as we discuss in Section 7.

4 | CHALLENGES FOR IMPLEMENTING THE APPROACH IN PRACTICE

The abstract approach presented in the previous sections has a number of assumptions on FaaS systems which are not necessarily true in existing systems. In this section, we discuss challenges for implementing the abstract approach in real-world FaaS systems.

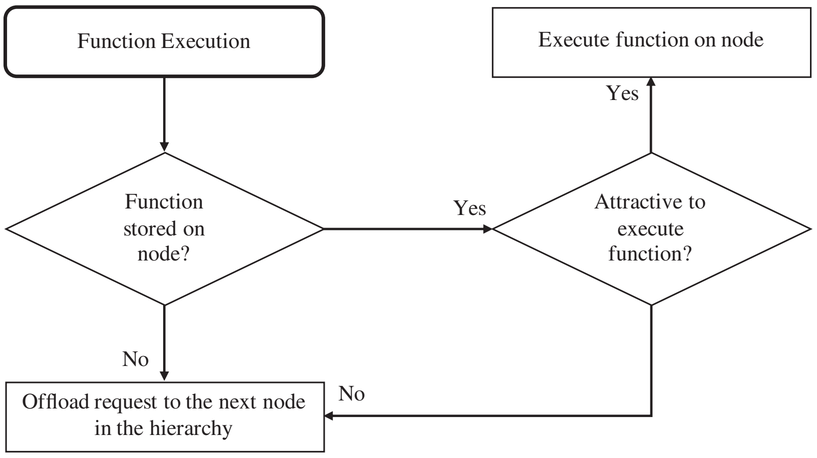

Function Execution
Function stored on node?
Yes
Attractive to execute function?
Yes
Execute function on node
No
Offload request to the next node in the hierarchy
No

FIGURE 5 Function execution process for a single node and a single function execution. We assume that the cloud stores and executes all functions with bids beyond a minimum price

We identified four challenges (C1–C4) for practically implementing our idea of an auction-based function placement in real-world FaaS systems. We present each of the challenges first in general, before we discuss the respective challenge using OpenWhisk as an example, as our proof-of-concept prototype (Section 6) is based on OpenWhisk.

4.1 | C1: Auctions arrive in batches

In our approach, nodes decide on the execution of specific functions by comparing the execution bids of requests. For this, we proposed a simple decision process where all incoming requests are ordered from high to low bids and the highest bid gets accepted as the execution set is limited by the required capacity. This process is sufficient as long as requests arrive in batches periodically. However, in many real-word scenarios requests will arrive arbitrarily and, therefore, it is not possible to directly compare the execution bids of all incoming requests. One approach to address this challenge, which we denote as *C1*, is collecting incoming requests for a certain time, effectively creating windows over the incoming stream of execution bids and, thus, the illusion of finite batches. The downside of this approach is that extra latency (up to the window length) is added to each request in each stage. Especially, requests that in the end are executed in the cloud will be affected by this extra latency several times. On the other hand, this approach is very simple to implement. It is even possible to use windowing only when the local resource utilization exceeds a predefined threshold, that is, when there is a chance of requests being offloaded. Another option is the use of statistical methods to estimate the arrival of incoming requests and height of bids, to use the available capacity for requests with relatively high bids based on the prediction model.

4.1.1 | OpenWhisk

In its architectural model, OpenWhisk separates load balancing from the part which limits the number of allowed function calls per minute. Therefore, the decision between execution and offloading has to be shifted to the decision-making part, while the Load Balancer employs strategies to select those functions that optimize the reward. To implement the two previously described methods, the Load Balancer could simply calculate an average execution bid over a certain time-frame. Assuming that the node has sufficient free capacities to accept a request, higher bids than the average get accepted while lower bids get offloaded.

4.2 | C2: Storage space calculation

Our approach assumes that the size of a function's executable is provided along with its storage bid. It also assumes that executables with lower bids get evicted to free up storage space whenever demand exceeds capacity. In practice, however, it is difficult to accurately estimate the storage space a function needs. Nevertheless, handling storage space well is important as the storage on edge devices is often significantly constrained. Allowing such nodes to store a certain number of functions might be a simple solution, yet the size of functions can differ considerably. For instance, a simple function may only need a few kilobytes (essentially, the cost for storing a few lines of code in a text file or in memory) while the size of rule-based classifier functions that bundle larger machine learning models may be in the high megabytes. Consequently, the number of functions is not directly suitable to constrain storage use. Instead, we need to consider the exact amount of required storage space for functions. Furthermore, most FaaS systems not only store the functions themselves, but also some data on executions and other metadata, where the size of this information grows over time and also needs to be taken into account. Arguably, the best approach may be to not only run the auction decision upon deploy time but rather to plan with a safety buffer in terms of storage space and to periodically rerun the auction over the set of locally stored executables and their metadata.

4.2.1 | OpenWhisk

OpenWhisk uses a database to store the functions and another one to store the metadata and output of successful executions. By setting a storage limit, each node can define the storage space available for functions: Since the exact function size in OpenWhisk is not known before the function is stored in the database, we have to ensure that the free storage space exceeds the maximum function size that we allow OpenWhisk to store by having a sufficiently large storage buffer[¶]. Moreover, we can use a second buffer to implement a limited storage for the metadata of previously executed functions. Once this buffer is full, the system can clean the history of execution data to ensure that the node can store data on further executions, for example, with an LRU eviction scheme. Thereby, the buffers ensure that enough free space is available to store the functions, past executions, and metadata.

4.3 | C3: Compute load estimation

Our approach entails that the FaaS nodes can execute a certain number of functions at the same time. In practice, however, one faces the challenge of having to estimate the supported number of concurrent function invocations. Setting the number of handled functions $\#\text{func}_n$ as shown in Equation 1, where mem_n is the node's available memory and memreq_i the memory requested by function $i \in \{1, 2, \dots, m\}$ (as in existing cloud FaaS deployment models), could lead to a simple result.

$$\#\text{func}_n = \frac{\text{mem}_n}{\frac{1}{m} \times \sum_{i=1}^{m} \text{memreq}_i}. \tag{1}$$

However, especially simple functions often do not fully use their reserved resources. In addition, most FaaS system support multiple programming languages and Docker containers, so that the resource usage of even similar functions can differ significantly. Therefore, FaaS systems can overbook the requested memory and often only distinguish between available and unavailable execution machines depending on the components' response times. In practice, we would suggest to couple the slight overbooking of memory resources with implicit CPU limits based on the number of local cores and the amount of compute capacity allocated to function instances.

4.3.1 | OpenWhisk

Instead of using a fixed amount of concurrent functions per Invoker, OpenWhisk distinguishes between healthy and unhealthy Invokers. However, when deploying OpenWhisk on more restricted hardware with capacities only for a limited

[¶]An OpenWhisk version released while we were writing this article changed this: OpenWhisk no longer checks whether the available storage exceeds the maximum function size. Instead, it checks whether the available storage size exceeds the actual function size plus an estimated buffer based on the base64 encoding in CouchDB. We decided not to change this in the article text as our prototype fork is based on the previous version and also since it does not really change the overall problem that the precise storage needs are unknown.

number of Invokers, it is important to have a good estimate for the function load. In addition, an unhealthy Invoker is identified as such after not responding for a certain time, yet this delay can be an issue for low-latency use cases, since latency can grow considerably before the Invoker is switched into an unhealthy state. Before simply distinguishing between healthy and unhealthy controllers, OpenWhisk used a method that checked the system load through active concurrent invocations in the system. This would arguably be a better strategy for low-latency use cases but may be problematic when resources are already constrained as on the edge.

4.4 | C4: Knowledge about the next node toward the cloud

Our approach assumes that nodes have knowledge about the next nodes toward the cloud. However, fog and IoT environments are often dynamic and latency between devices changes over time. In addition, nodes may fail and so can, correspondingly, any specific data transmission route toward the cloud. Therefore, practical implementations of our approach require a communication mechanism that allows forwarding to the cloud even in situations where the next node toward the cloud is not known and that is also robust against individual node failures. This could, for instance, be achieved by storing the full path to the cloud on every node rather than the identity of the next node.

4.4.1 | OpenWhisk

OpenWhisk was designed for deployment in a single cloud datacenter, not for fog environments. Therefore, offloading and shifting requests inside a network of OpenWhisk deployments is not part of OpenWhisk yet. In fact, it may be quite hard to implement since the information necessary for an offloading decision may be distributed over multiple components and machines of the OpenWhisk cluster. This means that the main challenge for OpenWhisk may be to transfer the abstract concept outlined above into a concrete implementation.

5 | EVALUATION: SIMULATION

In this subsection, we describe insights we gained from simulation. Namely, we ran three different simulation experiments: The first analyzes the effect of processing prices on function placement and latency depending on the request load. The second studies the effect of storage prices on function placement and latency depending on the demand for storage. The third experiment comprises a larger simulation in which we study how different executable replacement strategies, that are applied when running out of storage capacity, affect the storage and processing earnings of nodes.

5.1 | Simulation implementation

To evaluate our approach, we have implemented a simulation tool in Kotlin which is available on GitHub[#]. In the tool, we distinguish three types of nodes: edge, intermediary, and cloud. All three node types have distinct storage and processing capacities. Since we assume that all requests require the same compute power but may take arbitrarily long, the processing capacity of a node is specified as the number of requests which can be handled in parallel. The storage capacity of a node is specified as the number of function executables that the node can store on average. All nodes treat bids in the way described in Section 3. For future research on different auction strategies, it is only necessary to change the methods `offerExecutable` and `offerRequests` in ComputeNode.kt which implement the respective allocation rules (see Section 2).

Beyond the parameters already mentioned, users can specify the node setup (number and type of nodes and their interconnection), average latency for request execution as well as latency between edge and intermediary or intermediary and cloud, average storage and processing bids, the number and average size of executables, as well as the number of requests that should arrive at each edge node. The simulation comes preconfigured with a set of standard settings (defined in Configuration.kt) that can be changed to customize the simulation for various purposes. Our implementation also

[#]github.com/OpenFogStack/faas4fogsim.

assumes that bids are identical across all node types, that is, developers will bid and pay the same for execution and storage—no matter on which node.

To support reproducibility of simulation runs, the tool explicitly sets the random seed; parameters specified as average X follow a uniform distribution over a specified range.

5.2 | Configuration

For the first two simulation experiments, we had one edge, intermediary, and cloud node and simulated a period of two minutes; we also set the average function processing latency as 30 ms, the average edge to intermediary latency as 20 ms, and the average intermediary to cloud latency as 40 ms. For the third simulation experiment, we extended the topology: it comprised five edge nodes per intermediary, three intermediary nodes, and one cloud node, that is, 19 nodes in total.

In the first experiment, we set the storage capacity of each node to a very large value so that function execution was only delegated toward the cloud if a node did not have enough processing capacity available. All other settings remained as defined in the standard configuration. As a result, anything up to 166 req/s could on average be handled at the edge since the edge could process five requests concurrently. Furthermore, anything up to an average of 832 req/s could be handled at edge and intermediary since the intermediary could process 20 requests concurrently. Anything beyond this required the cloud for handling the load. Each request had a random processing bid $b_{processing} \sim \mathcal{U}_{[50,150]}$. We repeated this experiment with varying request load: from 100 to 10,000 req/s. For our 2 min (simulated time) experiments, we thus simulated 12,000 to 1,200,000 requests.

In the second experiment, we set the processing capacity of each node to a very large value so that function execution was only delegated toward the cloud if a node did not have enough storage capacity to store the function executable. All other settings remained as defined in the standard configuration. As a result, edge nodes could on average store 10 functions, intermediaries 50 functions, and the cloud an unlimited amount of functions. Executable sizes were randomly chosen as $size \sim \mathcal{U}_{[0.5,1.5]}$. Each executable also had a random storage bid $b_{storage} \sim \mathcal{U}_{[50,150]}$. We repeated this experiment with varying numbers of executables starting with 5 (which should still all fit on the edge node), over 50 (on average, edge nodes stored 20% of all executables and intermediaries still stored most of them), up to 100 in steps of 5.

In our third experiment, we evaluated the impact of our simple node strategy on function execution. As we have discussed, noncloud nodes make their decision about which function executables to accept or drop based solely on storage bids and do not consider the processing bid. We added a *stickiness* parameter to the simulation that describes the probability of a node ignoring an incoming function executable with a higher storage bid in favor of keeping its current set of function executables: A stickiness value of 0 corresponds to the default auction case from the other two simulation experiments, a stickiness value of 0.5 means that a bid which would normally evict other executables will do so with only a 50% probability, a stickiness value of 1 means that a node will never evict already stored executables (it will "stick" with the already stored ones). Please, note that we do not consider such a stickiness-based approach as a good strategy (as in "randomize if you follow the auction rules or not"). Instead, different strategies, for example, based on local preferences regarding certain users, will lead to different outcomes in terms of stickiness—hence, this is only a parameter to capture implications of a broad range of strategies as part of our simulation model, that is, variability of the sets of stored executables. For this experiment, we set the number of executables to 100. All other settings remained as defined in the standard configuration except for our updated topology (19 instead of three nodes). As a result, executable sizes were randomly chosen from $\mathcal{U}_{[0.5,1.5]}$, which means that edge nodes stored on average 10% and intermediaries stored on average 50% of all executables.

5.3 | Simulation experiment 1: Effect of processing prices

As can be seen in Figure 6A, the simulated system is lightly loaded up to about 400 req/s, since no requests are delegated toward the cloud. The intermediary begins to handle more requests than the edge at 300 req/s and the cloud begins to handle more requests than the intermediary at 1500 req/s.

For the earnings, we can see in Figure 6B that the average price for executing a function at the edge gradually increases as the edge node selects only the most lucrative requests for execution if it has a choice. For the intermediary, the average price remains slightly below 100 until 900 req/s, then it also starts to gradually increase. The cloud only processes its first

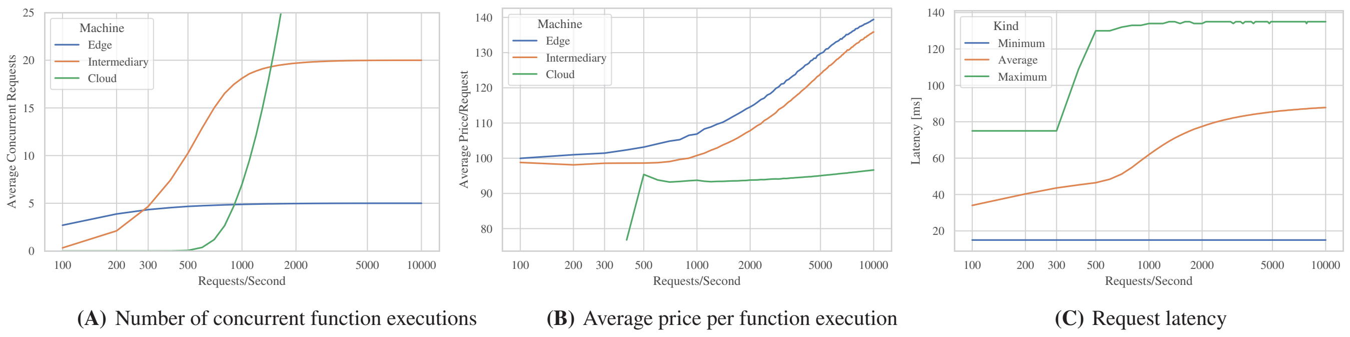


FIGURE 6 Simulation experiment 1: Studying the effect of varying request load when only execution capacities are auctioned

11 requests at 400 req/s. Since the intermediary only forwards the requests with the lowest processing bids, the cloud earnings are also quite low for this request rate.

For latency, we can see in Figure 6C that it linearly increases with the load as more and more requests (the lower paying ones) are no longer served on the edge but rather delegated to intermediary and cloud. We can also see the bump in maximum latency when the first request is executed in the cloud.

Overall, this experiment shows that even the simple auction scheme that we implemented is indeed an efficient mechanism to handle function allocation across fog nodes with the goal of determining the placement based on payment preferences of application developers.

5.4 | Simulation experiment 2: Effect of storage prices

In this experiment, we analyze what happens when the number of distinct functions increases (far) beyond the number of executables that can be stored on the edge and later even the intermediary. As can be seen in Figure 7A, the edge node begins to forwards requests to the intermediary at about 10 executables, while the intermediary begins to forwards requests to the cloud at about 50 executables (which is as expected based on our simulation parameters). The more functions, for which no executable is stored locally, are requested, the more requests are offloaded toward the cloud.

This is also visible from the average storage price shown in Figure 7B: the edge does not have any choice of executables when only five are offered so it also has to accept low storage bids. Likewise, the intermediary only starts to evict executables once it exceeds its capacity of about 50. Afterwards, both curves gradually increase toward the upper limit of 150 which, however, is not reached in this experiment as the randomization means that only 1% of the executables will be offered at a storage bid of 150. In addition, the average storage price in the cloud is about 100 which is as expected as the cloud accepts all executables in our simulation setting.

Figure 7C shows how an increasing number of executables also leads to increasing request latency: With every additional executable, the share of matching executables available at the edge (or on the intermediary) decreases so that requests are increasingly delegated toward the cloud since the function can, without an executable, not be executed.

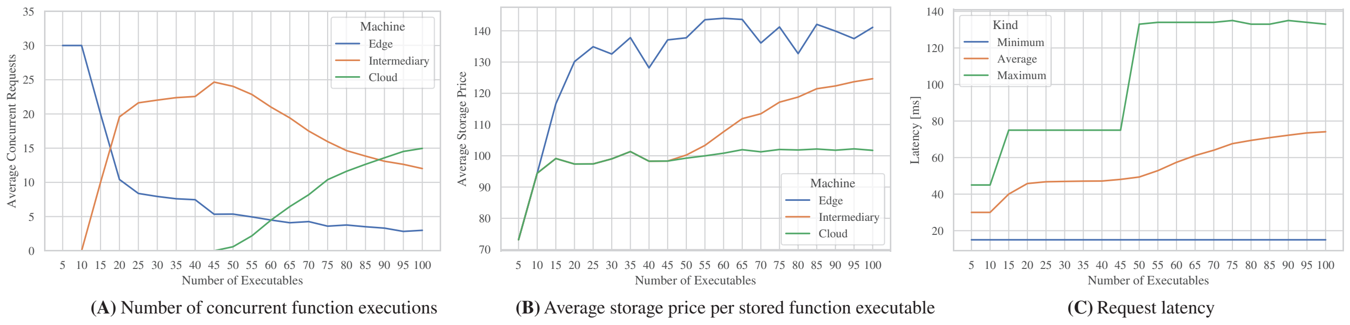
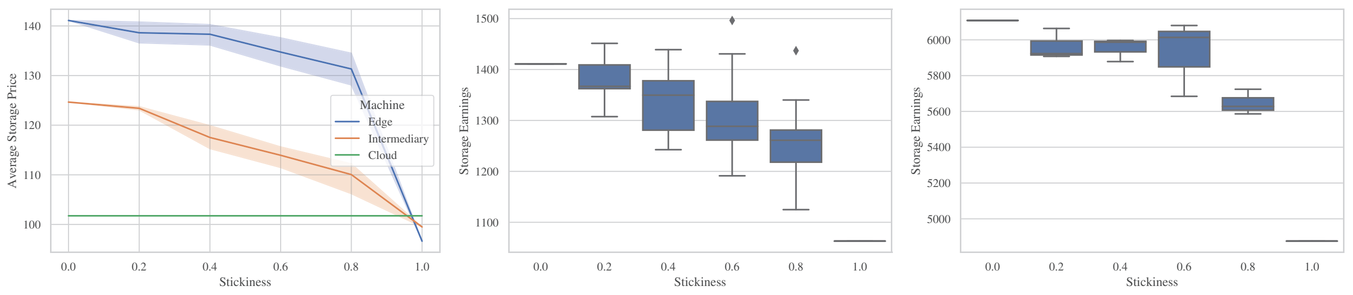


FIGURE 7 Simulation experiment 2: Studying the effect of varying number of executables when only the storage capacity is auctioned



(A) Average storage price per stored function executable **(B)** Distribution of storage earnings across edge nodes **(C)** Distribution of storage earnings across intermediary nodes

FIGURE 8 Simulation experiment 3: Studying the effect of varying degrees of stickiness in executable storage on storage earnings (one simulation run)

Overall, this experiment shows that our auction-based approach is an efficient mechanism to determine distribution of binaries across a fog network when there is a lack of storage capacity. Based on the experiment results, we would also argue that both bids should not be considered separately: From the perspective of an edge node, storing only the *n* executables that have the highest storage bid may not be optimal. For instance, if the *(n+1)*st executable (in terms of storage bid) is invoked significantly more often than the *n*th, storing the *(n+1)*st instead of the *n*th may offer higher earnings. The optimum, here, depends on request rates, execution bids, storage bids, and their respective distributions.

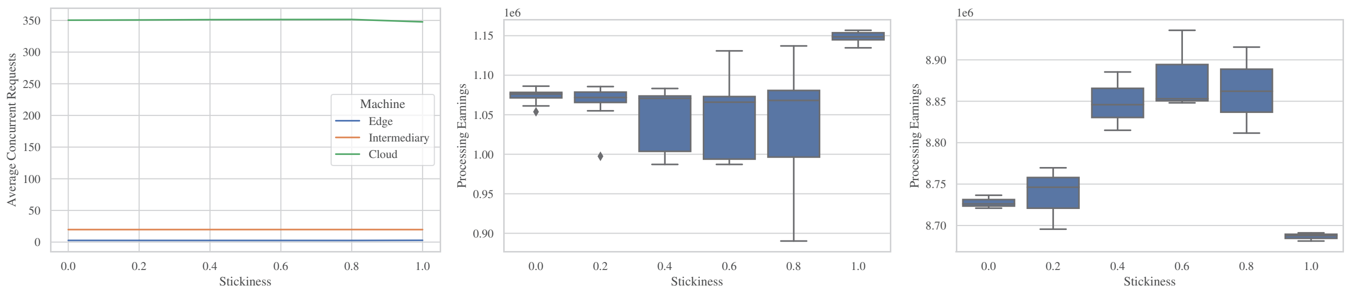
5.5 | Simulation experiment 3: Larger deployment and analysis of the effect of stickiness in executable management

In this experiment, we show that our simulation model is also suited for evaluating larger node topologies and analyze how different node strategies affect storage and processing earnings. As can be seen in Figure 8A, the average storage price for edge and intermediary nodes increases with a lower function stickiness. The reason for this is that a stickiness level of 0 means that the node will store the top X highest paying executable (in our example: on average the top 10 and top 50 out of 100 for edge and intermediary nodes respectively). A stickiness level of 1 means that the node will store the *first* X executables that are offered—essentially a random subset depending on the order in which storage requests come in[‖]. In general, it is therefore best to strictly follow our auction approach and to replace low-bid executables for maximizing the earnings from executable storage. There are, however, situations in which a node might benefit from not replacing an executable. For example, not replacing two small executables with a single large executable with a higher bid can result in higher total storage earnings. This can also be seen in Figure 8B which shows that some edge nodes achieved higher storage earnings when ignoring 20%–80% of storage requests compared to accepting all requests. Due to the larger overall storage capacity, this phenomenon does not matter for intermediary nodes in our simulation setup (Figure 8C).

Figure 9A shows that function stickiness has virtually no effect on the average number of concurrent requests carried out by each node; edge nodes have a load of 50% on average while intermediary nodes are fully loaded. The reason for this effect is that requests follow a uniform distribution across all executables. For stickiness levels of 0 and 1, all edge nodes (and intermediary nodes likewise) store the same set of executables. For values in between 0 and 1, there is some variance across nodes as some edge nodes (and intermediary nodes likewise) execute a few more requests. Due to the uniform distribution of requests, this variance is, however, so low that the curves which show it (using the same formatting as Figure 8A) are not visible in the chart.

Furthermore, Figure 9B shows that the median processing earnings are very similar for all nodes that periodically ignore storage requests since there is no correlation between processing and storage bids in our simulation setup. A higher function stickiness can lead to higher earnings for edge nodes, albeit at increased risk, as it depends on the processing bids for the two executables which are supposed to be swapped (or not). The biggest gamble for an edge node is a stickiness of 1: we repeated the experiment with different seeds and the processing earnings for such edge nodes can be very high

[‖]In our simulations setup, storage requests were sent in ascending order ordered by bid. Hence, the stickiness level of 1 results in the minimum possible storage earnings for the respective node.



(A) Number of concurrent function executions (B) Processing earnings for edge nodes (C) Processing earnings for intermediary nodes

FIGURE 9 Simulation experiment 3: Studying the effect of varying function stickiness on processing earnings (one simulation run)

or low compared to all other outcomes. The reason for this is that a stickiness level of 1 essentially picks a random subset of executables which each have random processing bids, thus, resulting in fully random processing earnings.

The low variability levels for both stickiness levels of 0 and 1 in Figure 9B,C, where all nodes of the respective type store the same executable set, result from the variability of requests across different nodes. It can be expected to asymptotically converge to 0 for very long running simulation experiments due to the uniform distribution of requests across executables and nodes.

Figure 9C shows that for intermediary nodes it is a beneficial strategy to choose a stickiness parameter between 0 and 1. The reason for this is that, in our setup, intermediary nodes maximize the probability of storing a different set of executables than their corresponding edge nodes. This means that requests offloaded by the edge due to a missing executable are more likely to find that executable on the intermediary (in the best case in our setup around 50% rather than 40% probability), that is, the executable has a choice from more requests and can achieve higher processing earnings. This effect is less pronounced when edge nodes are overloaded.

Overall, this shows that nodes should consider both processing and storage bids in an optimal strategy. They should also consider the frequency of execution requests for different executables. Finally, intermediary nodes should closely monitor the strategies of their edge nodes (or other intermediary nodes which are child nodes of them) and adapt correspondingly. For this, it may suffice to pretend that their respective child nodes are end devices sending requests and, otherwise, follow the same workload prediction strategies as edge nodes.

6 | EVALUATION: PROTOTYPE AND EXPERIMENTS

In this section, we present our proof-of-concept prototype AuctionWhisk which implements our auction-based approach in the open source FaaS platform (Lean) OpenWhisk. We then evaluate our approach through a number of experiments with AuctionWhisk.

6.1 | Prototypical implementation of AuctionWhisk

Our AuctionWhisk prototype is based on OpenWhisk and also supports the Lean OpenWhisk deployment type, which we refer to as Lean AuctionWhisk. Our auction-based function placement approach consists of two rounds, we therefore decided to split the description of our implementation into two parts: First (Section 6.1.1), we describe how to handle the storage bids and the resulting auction round. Second (Section 6.1.2), we present how we manage the execution of functions based on auction bids. We have made AuctionWhisk available as open source[**].

6.1.1 | Storing function executables

Before a function can be called, the executing node has to store the executable locally. OpenWhisk relies on CouchDB for storing function executables[††]. In this context, there are two relevant CouchDB databases: `whisk_local_whisks`

[**]github.com/OpenFogStack/AuctionWhisk.
[††]OpenWhisk refers to functions as *actions*.

stores the function executable and `whisk_local_activations` stores information about successful executions, including output and metadata. In AuctionWhisk, we extended the data model of `whisk_local_whisks` to also store the bids along with the executables and adapted the components interacting with this data model to also pass on the additional data.

In Section 4.2, we discussed challenge C2 and possible solutions for how nodes can estimate the required storage. In AuctionWhisk, each node can define the storage space available for functions by setting a configuration parameters for the perfunction storage limit as well as the total storage space available to AuctionWhisk. Both limits include all metadata of executed functions in `whisk_local_activations`which therefore should be garbage-collected periodically.

As a limitation of the original OpenWhisk, the actual storage size of a function is unknown. This also means that the Controller, which decides on accepting or rejecting a function executable, cannot directly know the actual amount of used storage. To circumvent this problem, we decided to use the perfunction storage limit (which by default in OpenWhisk is 48MB) as a proxy for the actual function size. Therefore, when AuctionWhisk is offered a new function executable, the system calculates the currently used storage as the number of stored executables times the perfunction storage limit and compares it to the total storage limit. If the estimated free space is larger than the perfunction storage limit, AuctionWhisk accepts the new executable and inserts it into the database. If not, AuctionWhisk retrieves the executable with the lowest storage bid and compares the bid to the new executable's bid. The one with the higher bid is then stored.

While this means that we reserve the per-function storage limit for every function, even for those that are possibly only a few KB in size, we believe that this is an elegant solution to the problem: The decision of which executables to store can easily be made and storage needs to be planned with a safety buffer anyhow since any execution of the function will increase the amount of stored data and functions may also need temporary disk space while executing. For a production environment, one could also analyze the actual function size after the deployment and use that for calculating the currently used storage for later requests. We decided not to do this in our prototype as this would require significant modifications to the OpenWhisk code base.

6.1.2 | Executing function calls

Upon invocation, OpenWhisk has to decide whether to execute or to offload the request based on the execution bid. This challenge is discussed in Section 4.4 as C4, where we assume that the nodes have knowledge about the next node toward the cloud. Once a request arrives at our node, the system checks whether the function executable is available locally. If this is not the case, standard OpenWhisk would return an error. In AuctionWhisk, however, the request is in that case forwarded to the next node on the path toward the cloud. The downside of this approach is that requests that target a function that does not exist anywhere in the system will always get their (high-latency) error message from the cloud. We believe, however, that this is acceptable since (i) we expect this scenario to be rare and (ii) the benefit of not having to store information on all functions on every FaaS node far outweigh this disadvantage. For instance, FaaS nodes in the US do not have to be aware of functions relevant only in Australia. We implemented this offloading behavior as part of the Controller and forward the request directly to the next node—ideally, an external messaging system would take care of delivering the request to the next *available* node on the path to the cloud.

As discussed with C3 (Section 4.3), it is relatively hard to estimate the available compute capacity, that is, how many additional requests can be accepted. Since OpenWhisk already uses the number of concurrent function executions to estimate CPU load, we have decided to adapt this approach in AuctionWhisk. We introduce a configurable parameter that allows users to define a time window length as well as the number of parallel requests that can be accepted in that time window. This is similar to a previous implementation in OpenWhisk but can be decided in the load balancer part of the Controller as it does not require information from all invokers of the node. If the request rate and the kind of function requested stays relatively constant, this approach allows a platform operator to find a good combination of configuration parameters.

If the average duration of requested functions increases, requests will be queued briefly; if it decreases requests will be offloaded even though there is still capacity left. On the one hand, this solution is simple to implement as it only requires some modifications to the existing code base. It also allows for a more fair and fine-grained scheduling of resources by the operating system scheduler as processes can requests all resources they need. On the other hand, it also increases the effects of "noisy neighbors", that is, breaking the isolation of function containers.[20] An approach to restrict CPU cores or even cycles is also feasible using Docker,[36,37] but would reverse that trade-off.

TABLE 1 System hardware configurations used for conducting the experiments

	Edge node	Intermediary node	Cloud node
Description	CPX21	CPX41	CPX51
CPU	3 vCPU	8 vCPU	16 vCPU
Memory	4 GB RAM	16 GB RAM	32 GB RAM
Network	10 GBit	10 GBit	10 GBit
Storage	NVMe SSD	NVMe SSD	NVMe SSD
OS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS	Ubuntu 18.04 LTS

As discussed in C1 (Section 4.1), the auction-based approach suffers from the problem that requests will not arrive in batches unless the system is completely overloaded. This means that AuctionWhisk will get individual requests and then has to decide whether to execute them or not. We have implemented two different approaches for this: In the first, the Controller simply creates microbatches by queuing requests in a short time window and then running an auction round over them. This is as close as possible to the vanilla auction approach which we described in Section 3. The downside of this is that every request on average gets half the time window length as additional latency which will only be acceptable if the execution duration of requested functions is much greater than the time window length.

The second approach which we have implemented (and which we will later use in the experiments) avoids this extra latency and for this slightly deviates from the auction approach. It introduces a new Decision Manager component within the Controller which uses a moving time window (default length one second) and tracks standard statistical aggregates such as average, median, min, max, and so forth. on the bids encountered in the time window (i.e., by default it knows the aggregates about all requests from the last second). Users can then define a custom function that calculates a threshold value based on these values—the default is the average. If there is enough capacity (as described in the paragraph above), requests whose bid exceeds that threshold will be accepted, all others will be offloaded. This is a bit different from the "pure" auction approach described in Section 3 but has the same or a very similar effect: When the average bid of requested functions stays constant, the threshold will adapt itself based on load patterns until the node executes only the highest paying functions and is fully utilized. When the average bid of requested functions increases, the threshold quickly adapts to the higher earning potential—it lags slightly behind depending on the window length but achieves at least the same earnings as in the constant scenario. When the average bid of requested functions decreases, the threshold also adapts quickly but may still reject a few requests near the threshold depending on the window length and how fast the average bid is decreasing. Overall, we believe that this is a good solution which can easily be implemented in practice but, of course, alternative implementations that actually predict future requests and thus achieve higher earnings are possible.

6.2 | Experiment setup

We also evaluated our approach through three experiments with AuctionWhisk. All three experiments used the same setup: The fog topology consists of one edge node, one intermediary fog node and one cloud node. The hardware configuration of the nodes can be found in Table 1. Please note that we used a relatively large edge node since OpenWhisk, the basis of AuctionWhisk, does not work well on smaller nodes.[10]

Since we did not have access to a real fog infrastructure, we injected artificial latency based on netEm[‡‡]similar to the approach used in MockFog[37,38]—see Figure 10 for an overview. Between the edge node and the intermediary node we set an average latency of 20 ms, the latency between the intermediary node and cloud node, we set to 40 ms, while our benchmarking client had a latency of 25 ms to the edge node (these numbers include both real and artificial delays). On the edge node, we installed Lean AuctionWhisk, the intermediary and cloud node ran standard AuctionWhisk.

As discussed above, AuctionWhisk uses two parameters to track available compute capacity: A moving time window and the number of requests which can be accepted in said time window. For the experiments, we set the time window length to 1000 ms and limited the number of requests for that time window to 6, 12, and 1000, respectively

[‡‡]man7.org/linux/man-pages/man8/tc-netem.8.html.

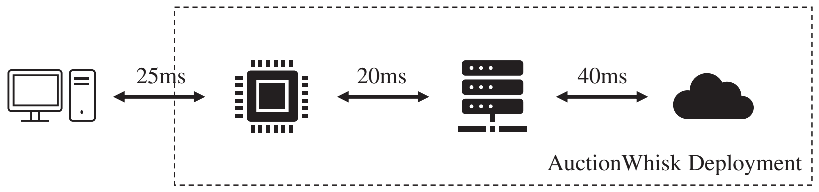


FIGURE 10 Experiment setup: Average network latency from client devices (on the left) via edge and intermediary fog nodes to the cloud

(edge/intermediary/cloud). We also increased the corresponding vanilla OpenWhisk parameters: For `invocationsPerMinute` we used 20,000 instead of 60 and for `concurrentInvocations` we used 240 instead of 30 to ensure that our experiments ran without failure. Without changing these parameters, only 60 invocations per minute and 30 concurrent invocations would have been allowed. These parameters were determined based on a number of initial experiments since standard OpenWhisk often has trouble coping with large numbers of parallel requests[10] and we wanted to avoid frequent error responses in our experiments.

As experiment workload, we used a Node.js 10 implementation of the sieve of Eratosthenes algorithm,[39] called with the argument 100, which was then automatically distributed over all nodes with a our deployment tool. For creating load, we used Apache JMeter[§§]and configured it to have several thread groups with several threads each which invoke the deployed functions for a period of 180 s. As the concrete thread/thread group configuration varied between the experiments, we point out the exact numbers in the respective sections.

6.3 | Experiment 1: Request latency of AuctionWhisk versus a cloud-only OpenWhisk

As the overall goal of the AuctionWhisk approach is to bring the benefits of edge and fog computing into the FaaS world, we ran a baseline experiment in which we compare request latency of AuctionWhisk to a cloud-only deployment of OpenWhisk.

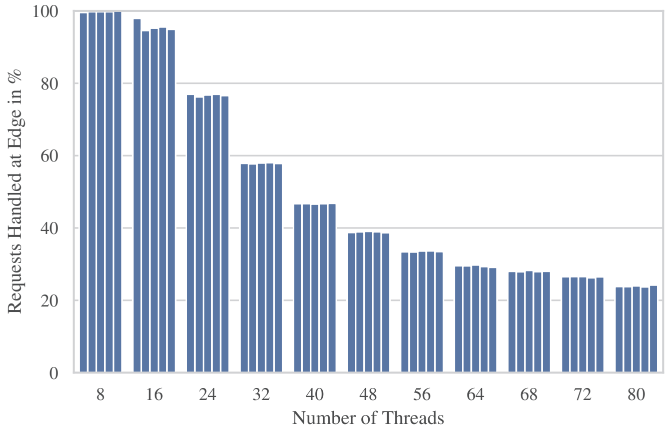
We used the standard setting from Section 6.2 and set our JMeter configuration to four thread groups. For each thread group, we started with two threads and increased that number in steps of two up to 20 threads per thread group (80 threads in total). Each thread issued a request every three seconds. We deployed one function per thread group and assigned it a random bid (uniformly distributed over the interval [50;150]). As a warm-up phase, we overloaded the target system for a short time to warm-up containers and to avoid cold starts during the actual experiment.[19,40] All experiments were repeated five times.

Figure 11 shows the results of all runs of experiment 1. With an increasing number of threads, as more and more requests are offloaded toward the cloud (Figure 11A), the median latency increases (Figure 11B). The first configuration with eight concurrent threads yields a latency of approximately 45ms, while all requests are handled at the edge. The configuration with 16 threads shows that the median latency is only slightly higher. However, the plot indicates that the edge slows starts offloading requests toward the intermediary node. At 24 threads, the edge starts offloading a larger percentage of requests to the intermediary node, which leads to a higher 75th percentile and a higher whisker. Running 40 threads, one can see that the median increases considerably. This is due to the fact that at this point the edge offloads more than 50% of all requests. Further, we observe a second change in the 75th percentile once there are 64 threads running: At this time, the intermediary starts offloading a large percentage of requests to the cloud. The outliers from the configuration with 56 threads indicate that single requests get offloaded at this point.

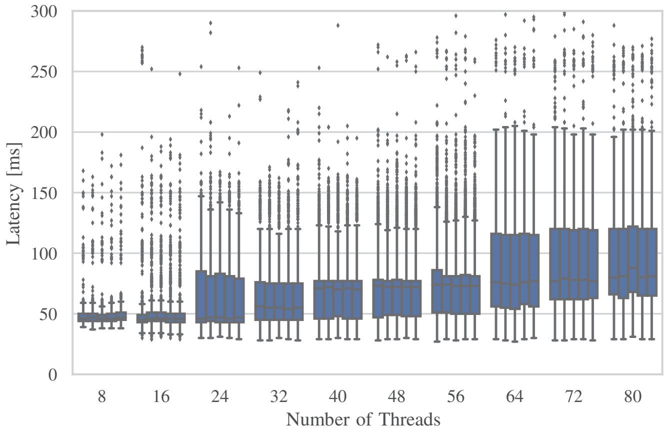
The CDF plot in Figure 12 also confirms these observation: The line for the experiment with eight threads shows a high slope and then plateaus close to value of 1—this indicates that almost all requests are processed on the edge. For all other configurations, we can see two "plateaus" (with a positive but low slope) which show the load levels which can still be sustained on one tier before offloading toward the cloud[¶¶].

For comparison, we also ran a cloud-only deployment with standard OpenWhisk in which we used a machine large enough to sustain the load level resulting from 64 threads. To assert a fair comparison (see Figure 10), we artificially

[§§]jmeter.apache.org.
[¶¶]If we had used n tiers instead of three tiers in our experiment, we would have expected $(n - 1)$ "plateaus".



(A) Impact of ofincreasing load on the percentage of requests that the edge can handle



(B) Impact on request latency as the load for AuctionWhisk is increased

FIGURE 11 Experiment 1: Increasing the load for AuctionWhisk gradually increases request latency as the percentage of requests handled at the edge decreases and more requests are offloaded to the cloud. At approximately 16 threads, the edge node is saturated and starts offloading; at approximately 56 threads, the intermediary node starts offloading to the cloud node

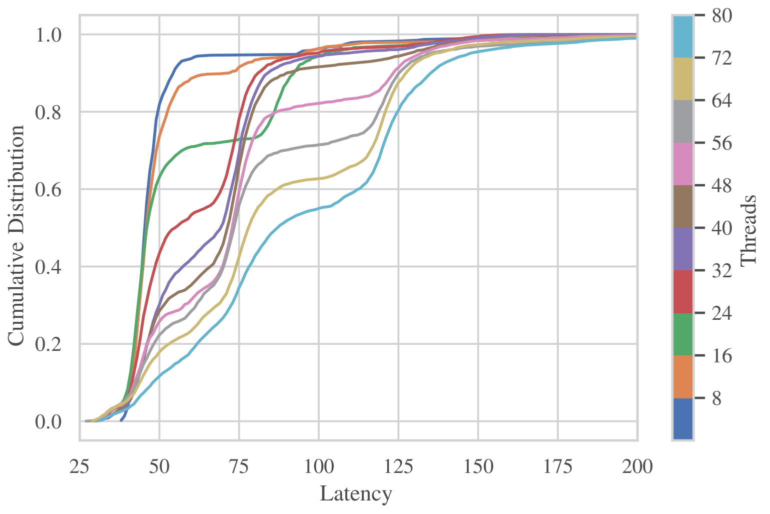


FIGURE 12 Experiment 1: CDF of AuctionWhisk runs at different load levels. The steep incline after "plateaus" indicates that any additional load is offloaded toward the cloud

delayed requests by 85 ms (=25 ms + 20 ms + 40 ms) and ran the exact same workload as in the AuctionWhisk experiments with 32 and 64 threads. As we can see in Figure 13, AuctionWhisk serves requests at least as fast as the cloud-only deployment of OpenWhisk##. While this is as expected, it shows that the AuctionWhisk approach allows OpenWhisk to efficiently leverage resources at or near the edge.

6.4 | Experiment 2: Effect of storage prices

The setup of this experiment is very similar to simulation experiment 2 from Section 5: We analyze the effect of storage bids on function placement and design the experiment in a way that asserts that the processing bids do not affect the outcome. In the experiment, we deployed the exact same function 1000 times with a random bid from the interval [50;150].

##The CDF does not reach the level of 1 in our chart as OpenWhisk often suffers from latency spikes and failed requests.[10]

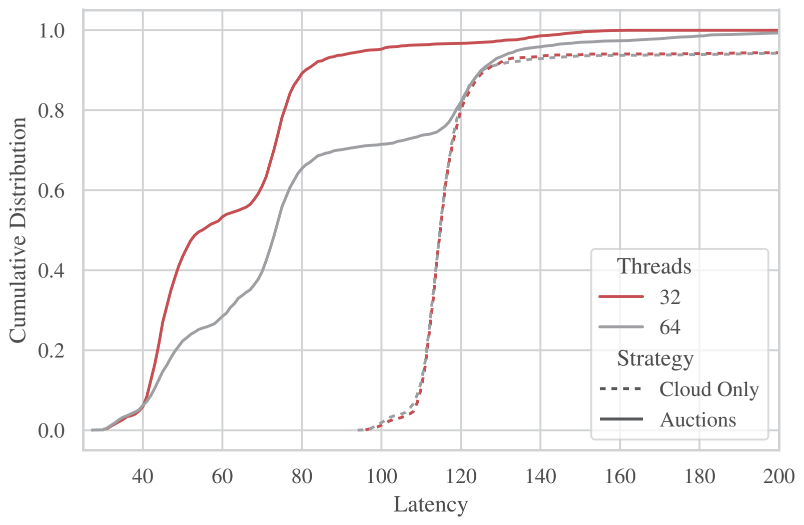
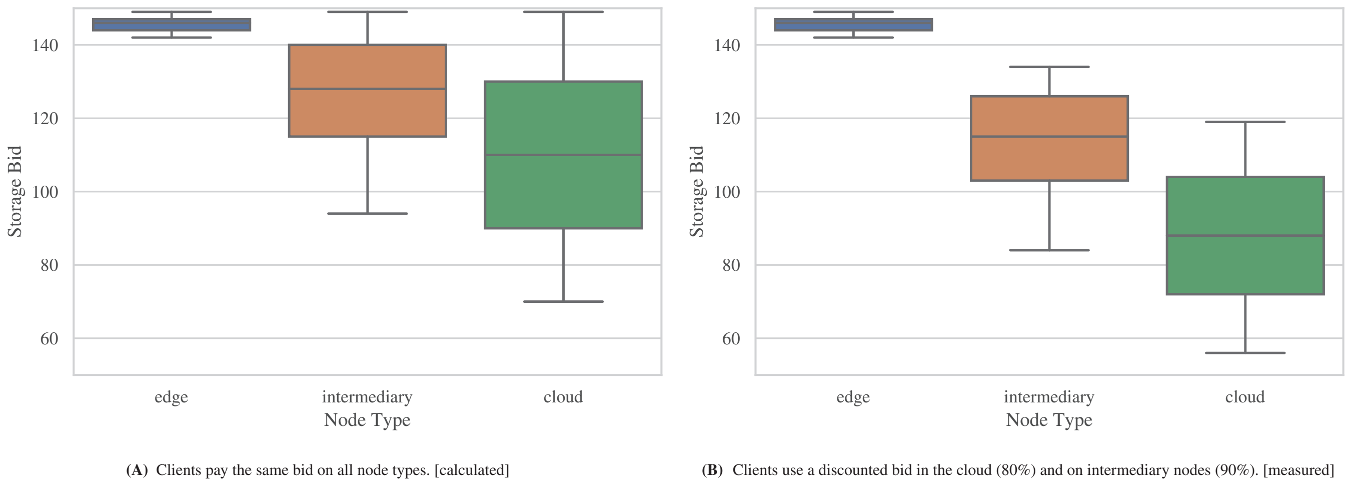


FIGURE 13 Experiment 1: CDF of AuctionWhisk and a cloud-only deployment of OpenWhisk for two different load levels. AuctionWhisk is at least as fast as OpenWhisk

To study storage bid effects in an isolated way, we artificially limited the storage capacity of the edge and intermediary node to assert that they could store only 100 and 500 functions respectively. In practice, developers can be expected to use higher bids for the edge or the intermediary (to increase the chance of their function executing there). For this experiment, we implemented this as a discount: functions with a bid b will pay b to edge nodes, $0.9 \cdot b$ to intermediary nodes, and $0.8 \cdot b$ to the cloud. We used detailed logging so that we could also calculate the node earnings without the discounted bids and repeated the experiment three times. Through previous test runs, we had already asserted that requests are indeed offloaded when an executable is not found locally and therefore decided not to run a workload against this deployment.

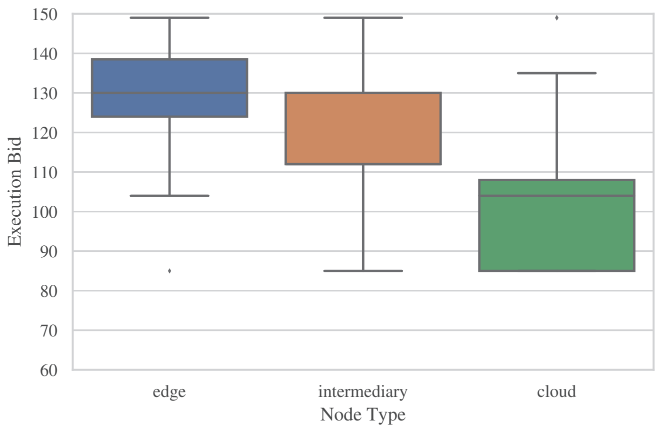
Figure 14 shows the results of one experiment run with (Figure 14B) and without (Figure 14A) discounted bids. As expected, we can see that functions stored at the edge have an average storage bid close to the upper bound. This is due to the restricted storage space, which forces the node to remove all functions with lower storage bids. In contrast to this, the cloud node stores all functions, from high paying to low paying—this can be clearly seen in Figure 14A where the upper whiskers for all node types have the same value. In general, both figures demonstrate the desired effect while the discounted bids (which could also be implemented as a surcharge on the edge) further aggravate the effect. The exact same effect can be seen in the other two test runs which, however, had slightly different values due to the randomization of bids.



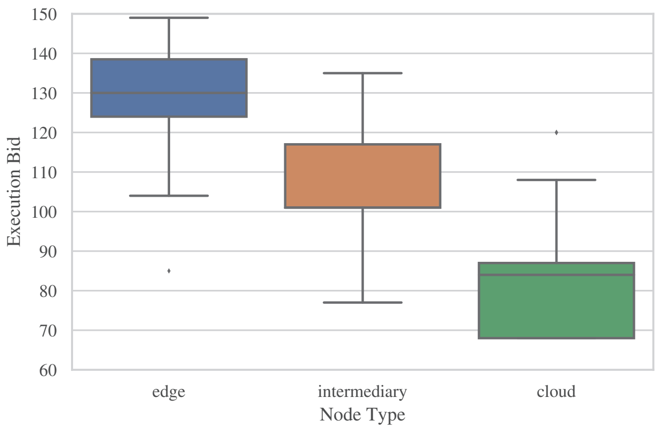
(A) Clients pay the same bid on all node types. [calculated]

(B) Clients use a discounted bid in the cloud (80%) and on intermediary nodes (90%). [measured]

FIGURE 14 Experiment 2: Distribution of successful bids on different node types. Nodes with limited capacity at or near the edge only store the executables with the highest bids while the cloud stores all executables



(A) Clients pay the same bid on all node types. [measured]



(B) Clients use a discounted bid in the cloud (80%) and on intermediary nodes (90%). [calculated]

FIGURE 15 Experiment 3: Distribution of prices paid per execution across different node types. The higher the processing bid of the requested function is, the more likely is the request to be executed at or near the edge

6.5 | Experiment 3: Effect of execution prices

The setup of this experiment is very similar to Simulation Experiment 1 from Section 5: We analyze the effect of processing bids on function placement and design the experiment in a way that asserts that the storage bids do not affect the outcome. We deployed the same function 100 times, each with a random processing bid in the interval [50;150]. For the workload, we used eight parallel thread groups (with eight threads per thread group) which each targeted one randomly selected function, that is, we invoked eight different functions which each had their respective processing bid. We repeated the experiment three times.

The results of one experiment run are shown in Figure 15: similar to experiment 2, we not only show the results of the base setup as described above (Figure 15A) but also calculated the outcome for developer discounts on intermediary nodes (90% of the edge price) and in the cloud (80% of the edge price)—see Figure 15B. As expected, we can see that the distribution of prices paid for execution strongly increases toward the edge as the nodes with limited resources are able to cherry-pick the highest paying requests only. With the discount option, this effect is even more apparent. The other two experiment runs confirm these results but have slightly different values due to the randomization when selecting the 8 out of 100 target functions.

Overall, our experiments with AuctionWhisk show the desired effect of an efficient resource allocation where the location of a function execution is determined by the price the application developer is willing to pay.

7 | DISCUSSION

With our simulation models and empirical measurements on our prototype AuctionWhisk, we have shown that it is feasible to place functions in fog-based FaaS platforms using decentralized auction-inspired mechanisms. In the following, we will discuss limitations of our approach and derive avenues for future work.

Bidding of application developers. In our experiments, we assume simple bidding strategies from application developers, yet more advanced strategies, such as developers providing high bids for scarce resources that are urgently needed while offering low bids for, for example, processing and storage in the cloud, are also possible.

Although this is not the focus of our research, our auction-inspired approach also invites a game-theoretic perspective, for example, on combining storage and processing bids such as to use a high storage bid with a bloated executable to effectively get a single tenant edge node combined with a very low processing bid to reduce cost.

Furthermore, as bids are provided at deploy time, they are relatively static. This means that developers cannot easily take advantage of periods of low demand (e.g., at night) with temporarily lower bids and reduced overall costs. Also, actual prices may vary a lot over time depending on the actions of other developers as well as the request rates created by end users. The result is that earnings at the edge are maximized, with the goal of making the high CapEx of installing edge infrastructure worthwhile, while tenants can specify how much they are willing to pay.

As an additional factor, the impact of providing information about the location of function execution, system load, or bids of other tenants on optimal bidding strategies may be an interesting subject for further research.

Auction mechanism. In our simulation model, nodes are rather simplistic in that they do not look into the future and do not couple storage and processing bid—they simply accept all executables that fit in and serve whatever is received in the order of the bids. While this is in line with our chosen auction model, much higher revenue could be achieved through decisions that follow pricing trends. For instance, just considering the processing bid together with past invocation numbers and coupled with the storage bid could result in considerably higher profits. As a more concrete example, it may make sense to delegate a processing request even though the node is not saturated when a periodic request with a higher processing cost is expected to arrive shortly.

Likewise, our AuctionWhisk prototype considers both bids separately through the heuristics described in Section 6.1. We decided not to use the microbatching approach for the execution auction in our experiments as it increases the overall latency significantly. Nevertheless, microbatching would further increase the earnings for the respective node. For a deployment in practice, we suggest to use microbatching when the latency to the next node on the path to the cloud is high and our proposed heuristic in all other cases.

Overall resource allocation. Combining both sides of the auction, significant improvements are possible when application developers try to minimize cost while nodes try to maximize earnings. However, even then the overall solution will not yield the globally optimal result that could be achieved through an allocation optimization across all nodes. Yet, in exchange for optimizing only locally, nodes do not need to communicate for their decision, leading to faster decisions and thus lower end-to-end latency, more overall scalability, and potentially also an increased resiliency against connection losses. In some scenarios, it may also be beneficial not to offload a request but rather to queue it on the edge briefly (thus possibly reducing end-to-end latency while keeping the earnings for the edge node).

No guarantees for application developers. Overall, our solution takes a rather particular approach in that application developers get no guarantees where their function will be executed—the placement of functions only depends on the bids at a specific point in time. There is, hence, no way to provide bounds on response time and other quality dimensions, that is, the infrastructure provider does not provide any guarantees beyond *eventually* executing every function *somewhere*. In a way, this is comparable to airline seat assignment schemes in which economy class passengers can bid for business class upgrades. While this is undesirable from an application developer perspective, our proposed approach has the benefit that it requires no centralized coordination at all, that is, it scales well and is resilient, and that it is likely to maximize earnings for the infrastructure provider. Also, the lack of guarantees is in line with the state-of-practice in cloud services which provide minimal to no guarantees as part of their SLAs.[20,41]

Still, if our proposed approach were to be used in practice, we would expect infrastructure providers to let customers reserve subsets of their resources at a fixed price—comparable to 5G slicing[42]—and use the proposed auctioning scheme only for a subset of their resources. This would lead to the interesting situation that application developers that actually need quality guarantees can pay for it (comparable to directly booking a business class seat) while others can still bid on the remaining resources (comparable to bidding on a business class upgrade).

Finally, developers can currently not get a guarantee that function execution happens only in a certain area to, for example, satisfy regulatory or privacy requirements. While they could decide to send their executable to only a limited subset of nodes, requests that are outbid on the edge will still be propagated to the cloud before the cloud node detects that it does not have an executable for the request. With personal data of end users governed by the GDPR, this might be illegal depending on the location of the cloud node. One way to address this would be to let developers not only specify bids but also constraint individual offloading paths of functions. This way, a request could be offloaded only to desired nodes (possibly nearby edge nodes).

Implementation challenges. There are of course a number of technical challenges in implementing a fully integrated platform that we did neither address with our concept nor our prototype implementation. For instance, we did not consider node failures or network partitioning: The former case could be addressed by using a reliable messaging middleware for communication between nodes, which would ensure that all requests are eventually delivered to a node for execution. In the case of network partitioning, offloaded requests will be lost in our current prototype. We do, however, not see this as a major limitation as, in such a scenario, some requests *will have to* be dropped.

There is also the aspect of co-managing data placement[4,13,43,44] and function placement,[12,21,45,46] which we think is a promising avenue for future research.[8,47]

Finally, we did not consider undeployment of functions: Depending on storage demand, this may remove the executable from an otherwise fully utilized node. The node, however, then no longer knows of rejected or evicted executables, thus potentially reducing future earnings in this node. This problem could be addressed by having a repository where developers submit their executables and bids as a single source of truth that nodes could then query to gain access to executables even after evictions.

Summary. While there are some limitations in our approach, we believe its benefits outweigh the disadvantages. Most of the limitations could also be addressed through compensation mechanisms or small extensions. Furthermore, the overall AuctionWhisk approach opens up a range of interesting game-theoretic research questions, which are beyond the scope of this article but worthy of further research.

8 | RELATED WORK

The service placement problem is inherent to fog computing, and deploying distributed applications over a distributed, heterogeneous infrastructure under constraints such as bandwidth usage, latency limits, or regulatory requirements is a combinatorial problem. Consequently, this challenge has been a key research topic in the last few years. On the one hand, there are static solutions using upfront best practices,[12,48-50] simulation,[45,51-60] testbed evaluation,[38,61-67] a combination of those,[46,68] or formalized assignment problems.[69-79] On the other hand, dynamic approaches using centralized schedulers[80-91] or decentralized algorithms[92-95] have also been proposed. It is no surprise that auction-based approaches, which have seen broader interest in computer science,[28-35] have also been proposed for fog computing, especially in those multiprovider, multitenant scenarios where both infrastructure providers and application service providers want to find a cost-optimal solution to service placement.[96-121] For example, Fawcett et al.[96] present an auction-based resource allocation platform for the fog that has service providers interested in using the fog infrastructure bid for resources provided by independent infrastructure providers. They employ combinatorial bids for combinations of CPU, storage, and memory for a specified time slot and assume services to be available in the form of containers. A central orchestrator acts as an auctioneer toward service providers and collects their bids, then performs a provisioning step where it provisions the services of the winning bids on the available devices.

Nevertheless, existing research has not taken into account the unique challenges and opportunities of fog-based FaaS platforms, although it has been shown that the FaaS paradigm is a good fit for fog applications.[1,10,12,122] Notably, the finely-grained execution units of serverless functions can lead to high load on central scheduling components, as Rausch et al.[123] have shown in comparison to their earlier work.[124-126] The authors claim that there is no technical solution that is able to handle such scheduling of function execution across edge, intermediaries, and the cloud. On the other hand, precisely this flexible scheduling of function executions can maximize resource allocation and lead to better auction conditions for both providers as well as consumers. Using long-running services that are migrated with user movement (e.g., as suggested by References 127 and 128) blocks scarce resources on constrained and expensive edge and fog nodes and is thus less efficient than executing functions only when needed.

Nevertheless, some scheduling solutions using centralized monitoring or control units have been proposed: For example, Aske and Zhao[129] propose a central monitoring solution that collects telemetry data from different fog FaaS platforms and schedules functions to the best platform given perfunction constraints. However, this leaves the majority of scheduling work with the platform providers as it abstracts from the actual platforms. Cheng et al.[130] introduce function coordination based on "data," "usage," and "system" contexts to optimize data-intensive applications. To some extent, this assumes scalable infrastructure as resources need to be available where data is produced, and a central coordinator needs to keep track of all data streams. Furthermore, as this approach is tailored to data-intensive applications, it can probably not deal with event-driven applications. Pelle et al.[131] as well as Palade et al.[132] use centralized controllers for function scheduling, albeit not on a perexecution basis. Nevertheless, this requires a central, global view of the FaaS system, which is especially unfeasible in a multiprovider fog setup.

Decentralized approaches forego such a central component to increase scalability and availability. Persson and Angelsmark[133] build their *Kappa* platform on top of *Calvin*,[134] as distributed IoT platform. They deploy functions as applications on top of Calvin, yet it is unclear how function scheduling is achieved beyond static assignment of functions to compute nodes. Baresi and Medonca[21] propose a serverless edge platform that comprises independent FaaS platforms distributed across the fog. The platforms coordinate function offloading among each other if needed, based on current load and request QoS requirements. Although their approach avoids a centralized scheduler that could introduce a bottleneck or single point of failure, it is unclear on what basis function offloading is decided. Regional load balancers distribute requests within their region, yet the paper remains vague regarding the question whether multiple such load balancers exist or what their impact is on request latency, as additional network hops introduce overhead. In their earlier work,[135] clients themselves would need to specify whether to execute a function at a local edge node or in the cloud. This is also a common approach in commercial solutions for edge function execution. Most notably, AWS offers Greengrass[||||] and Lambda@Edge[***], while Microsoft has added Azure Stack Edge[†††]to their portfolio. These solutions require application developers to address function execution environments at the edge or in the cloud in different ways, so that the burden of resource management is left to developers. To this end, Das et al.[136] propose client-side models that predict whether an execution on the edge or in the cloud is more efficient to satisfy given constraints. Cicconetti et al.[137] argue for uncoordinated access, where clients are given a list of possible function execution locations and make local decisions, for example through probing different options and adapting over time. While this does not lead to a global optimum, the scheduling load on clients and infrastructure is low. Two assumptions make this a worse deal for infrastructure providers than our auction-based approach: first, all possible function nodes must store the function code at all times as they do not know when a client changes their decision on execution location. Second, an edge node cannot influence when it would like to be considered as an execution location for a function, for instance once it has enough resources available. This prohibits a dynamic pricing policy. The authors have also proposed using intermediary routers[138] as decentralized function schedulers, an approach where both clients and nodes have no control over function execution.

9 | CONCLUSION

Serverless FaaS is a promising paradigm for fog environments. In practice, fog-based FaaS platforms have to schedule the execution of functions across multiple geo-distributed sites, especially when edge nodes are overloaded. Existing approaches mostly argue for centralized placement decisions which, however, will not scale indefinitely.[123]

In this article, we extend a previous paper[9] and follow an auction-inspired scheme in which application developers specify bids for storing executables and executing functions across the fog. This way, overloaded fog nodes can make local decisions about function execution. Therefore, also the function placement is distributed across nodes and no longer the scalability bottleneck it is with centralized scheduling approaches. Furthermore, we showed through simulation that our approach asserts that all requests are served while maximizing revenue for overloaded nodes. We also showed that such an approach can be implemented in real-world FaaS platforms by extending the open source system Apache OpenWhisk for our AuctionWhisk prototype and conducting three experiments with it.

ACKNOWLEDGMENTS

The authors thank Setareh Maghsudi for contributing an overview of various auction approaches to our original paper.[9] Open Access funding enabled and organized by Projekt DEAL.

DATA AVAILABILITY STATEMENT

We make all our prototype implementations available as open-source software. Namely, this concerns our simulation toolkit (https://github.com/OpenFogStack/faas4fogsim) and our AuctionWhisk prototype (https://github.com/OpenFogStack/AuctionWhisk).

ORCID

David Bermbach https://orcid.org/0000-0002-7524-3256
Jonathan Hasenburg https://orcid.org/0000-0001-8549-0405
Tobias Pfandzelter https://orcid.org/0000-0002-7868-8613
Lauritz Thamsen https://orcid.org/0000-0003-3755-1503

REFERENCES

1. Bermbach D, Pallas F, Perez DG, et al. A research perspective on fog computing. Proceedings of the International Conference on Service-Oriented Computing. 2017:198-210; Málaga, Spain.
2. Bonomi F, Milito R, Zhu J, Addepalli S. Fog computing and its role in the Internet of Things. Proceedings of the 1st edition of the MCC workshop on Mobile cloud computing. 2012:13-16; Helsinki, Finland.
3. Lu R, Heung K, Lashkari AH, Ghorbani AA. A lightweight privacy-preserving data aggregation scheme for fog computing-enhanced IoT. *IEEE Access*. 2017;5:3302-3312.

[||||]aws.amazon.com/greengrass.
[***]aws.amazon.com/lambda/edge.
[†††]azure.microsoft.com/en-us/products/azure-stack/edge.

4. Zhang B, Mor N, Kolb J, et al. The cloud is not enough: saving IoT from the cloud. Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15); 2015; Santa Clara, CA.
5. Wiesner P, Thamsen L. LEAF: simulating large energy-aware fog computing environments. Proceedings of the 2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC); 2021:29-36; Online, Australia.
6. Pallas F, Raschke P, Bermbach D. Fog computing as privacy enabler. *IEEE Internet Comput*. 2020;24(4):15-21.
7. Baldini I, Cheng P, Fink SJ, et al. The serverless trilemma: function composition for serverless computing. Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software; 2017:89-103; Vancouver, BC.
8. Hellerstein JM, Faleiro J, Gonzalez JE, et al. Serverless computing: one step forward, two steps back. Proceedings of CIDR' 19; 2019; Asilomar, CA.
9. Bermbach D, Maghsudi S, Hasenburg J, Pfandzelter T. Towards auction-based function placement in serverless fog platforms. Proceedings of the Second IEEE International Conference on Fog Computing (ICFC 2020); 2020:25-31; Sydney, NSW, Australia.
10. Pfandzelter T, Bermbach D. tinyFaaS: a lightweight FaaS platform for edge environments. Proceedings of the Second IEEE International Conference on Fog Computing (ICFC 2020); 2020:17-24; Sydney, NSW, Australia.
11. Agache A, Brooker M, Iordache A, et al. Firecracker: lightweight virtualization for serverless applications. Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20); 2020:419-434; Santa Clara, CA.
12. Pfandzelter T, Bermbach D. IoT data processing in the fog: functions, streams, or batch processing? Proceedings of the 2019 IEEE International conference on fog computing (ICFC); 2019:201-206; Prague, Czech Republic.
13. Confais B, Lebre A, Parrein B. An object store service for a fog/edge computing infrastructure based on IPFS and a scale-out NAS. Proceedings of the 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC); 2017:41-50; Madrid, Spain.
14. Baldini I, Castro P, Chang K, et al. *Serverless Computing: Current Trends and Open Problems*. Springer; 2017.
15. Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Serverless computation with openLambda. Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16); 2016; Denver, CO.
16. Lynn T, Rosati P, Lejeune A, Emeakaroha V. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom); 2017:162-169; Hong Kong SAR.
17. Akkus IE, Chen R, Rimac I, et al. SAND: towards high-performance serverless computing. Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18); 2018; Boston, MA.
18. The Apache Software Foundation. OpenWhisk documentation. Accessed May 8, 2021. https://openwhisk.apache.org/documentation.html
19. Bermbach D, Karakaya AS, Buchholz S. Using application knowledge to reduce cold starts in FaaS services. Proceedings of the 35th ACM Symposium on Applied Computing (SAC 2020); 2020:134-143; Brno, Czech Republic.
20. Bermbach D, Wittern E, Tai S. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer; 2017.
21. Baresi L, Mendonça DF. Towards a serverless platform for edge computing. Proceedings of the 2019 IEEE International Conference on Fog Computing (ICFC); 2019:1-10; Prague, Czech Republic.
22. George G, Bakir F, Wolski R, Krintz C. NanoLambda: implementing functions as a service at all resource scales for the Internet of Things. Proceedings of the 2020 IEEE/ACM Symposium on Edge Computing (SEC); 2020:220-231; San Jose, CA.
23. Hall A, Ramachandran U. An execution model for serverless functions at the edge. Proceedings of the International Conference on Internet of Things Design and Implementation; 2019:225-236; Montreal, Canada.
24. Aslanpour MS, Toosi AN, Cicconetti C, et al. Serverless edge computing: vision and challenges. Proceedings of the 2021 Australasian Computer Science Week Multiconference. 2021:1-10; Dunedin, New Zealand.
25. Pfandzelter T, Hasenburg J, Bermbach D. Towards a Computing Platform for the LEO Edge. Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking; 2021:43-48; Online, UK.
26. Palade A, Kazmi A, Clarke S. An evaluation of open source serverless computing frameworks support at the edge. Proceedings of the 2019 IEEE World Congress on Services (SERVICES); 2019:206-211; Milan, Italy.
27. Mell P, Grance T. *The NIST Definition of Cloud Computing*. NIST Special Publication; 2011:800-145.
28. Huang J, Han Z, Chiang M, Poor HV. Auction-based resource allocation for cooperative communications. *IEEE J Select Areas Commun*. 2008;26(7):1226-1237.
29. Gao L, Xu Y, Wang X. MAP: multiauctioneer progressive auction for dynamic spectrum access. *IEEE Trans Mob Comput*. 2011;10(8):1144-1161.
30. Zhao D, Li X, Ma H. How to crowdsource tasks tastfully without sacrificing utility: online incentive mechanisms with budget constraint. Proceedings of the INFOCOM; 2014:1213-1221; Toronto, ON, Canada.
31. Yang D, Xue G, Fang X, Tang J. Incentive mechanisms for crowdsensing: crowdsourcing with smartphones. *IEEE/ACM Trans Netw*. 2016;24(3):1732-1744.
32. Kantarci B, Mouftah HT. Trustworthy sensing for public safety in cloud-centric Internet of Things. *IEEE Internet Things J*. 2014;1(4):360-368.
33. Zhang L, Li Z, Wu C. Dynamic resource provisioning in cloud computing: a randomized auction approach. Proceedings of the INFOCOM; 2014:433-441; Toronto, ON, Canada.
34. Chandrashekar TS, Narahari Y, Rosa CH, Kulkarni DM, Tew JD, Dayama P. Auction-based mechanisms for electronic procurement. *IEEE Trans Automat Sci Eng*. 2007;4(3):297-321.

35. Jayaweera SK, Bkassiny M, Avery KA. Asymmetric cooperative communications based spectrum leasing via auctions in cognitive radio networks. *IEEE Trans Wirel Commun*. 2011;10(8):2716-2724.
36. Symeonides M, Georgiou Z, Trihinas D, Pallis G, Dikaiakos MD. Fogify: a fog computing emulation framework. Proceedings of the 2020 IEEE/ACM Symposium on Edge Computing (SEC); 2020:42-54; San Jose, CA.
37. Hasenburg J, Grambow M, Bermbach D. MockFog 2.0: automated execution of fog application experiments in the cloud. *IEEE Trans Cloud Comput*. 2021.
38. Hasenburg J, Grambow M, Grünewald E, Huk S, Bermbach D. MockFog: emulating fog computing infrastructure in the cloud. Proceedings of the 2019 IEEE International Conference on Fog Computing (ICFC); 2019:144-152; Prague, Czech Republic.
39. O'Neill ME. The genuine sieve of Eratosthenes. *J Funct Program*. 2009;19(1):95-106.
40. Mohan A, Sane H, Doshi K, Edupuganti S, Nayak N, Sukhomlinov V. Agile cold starts for scalable serverless. Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19); 2109; Rento, WA.
41. Bradshaw S, Millard C, Walden I. Contracts for clouds: comparison and analysis of the terms and conditions of cloud computing services. *Int J Law Inf Technol*. 2011;19(3):187-223.
42. Foukas X, Patounas G, Elmokashfi A, Marina MK. Network slicing in 5G: survey and challenges. *IEEE Commun Mag*. 2017;55(5):94-100.
43. Hasenburg J, Grambow M, Bermbach D. Towards a replication service for data-intensive fog applications. Proceedings of the 35th ACM Symposium on Applied Computing, Posters Track (SAC 2020); 2020:267-270; Brno, Czech Republic.
44. Hasenburg J, Grambow M, Bermbach D. FBase: a replication service for data-intensive fog applications. Technical report; 2019.
45. Hasenburg J, Werner S, Bermbach D. Fogexplorer. Proceedings of the 19th International Middleware Conference (Posters); 2018:1-2; Rennes, France.
46. Pfandzelter T, Hasenburg J, Bermbach D. From zero to fog: efficient engineering of fog-based IoT applications. *Softw Pract Exp*. 2021;51(8):1798-1821.
47. Sreekanti V, Wu C, Lin XC, et al. Cloudburst: stateful functions-as-a-service. *Proc VLDB Endow*. 2020;13(12):2438-2452.
48. Gusev M, Koteska B, Kostoska M, et al. A deviceless edge computing approach for streaming IoT applications. *IEEE Internet Comput*. 2019;23(1):37-45.
49. Karagiannis V, Schulte S. Comparison of alternative architectures in fog computing. Proceedings of the 2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC); 2020:19-28; Melbourne, VIC, Australia.
50. Santos L, Silva E, Batista T, Cavalcante E, Leite J, Oquendo F. An architectural style for Internet of Things systems. Proceedings of the 35th Annual ACM Symposium on Applied Computing; 2020:1488-1497; Brno, Czech Repbulic.
51. Hasenburg J, Werner S, Bermbach D. Supporting the evaluation of fog-based IoT applications during the design phase. Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things; 2018:1-6; Rennes, France.
52. Gupta H, Vahid Dastjerdi A, Ghosh SK, Buyya R. iFogSim: a toolkit for modelling and simulation of resource management technologies in the Internet of Things, edge and fog computing environments. *Softw Pract Exper*. 2017;47(9):1275-1296.
53. Jha DN, Alwasel K, Alshoshan A, et al. IoTSim-Edge: a simulation framework for modeling the behavior of Internet of Things and edge computing environments. *Softw Pract Exper*. 2020;50(6):844-867.
54. Brambilla G, Picone M, Cirani S, Amoretti M, Zanichelli F. A simulation platform for large-scale Internet of Things scenarios in urban environments. Proceedings of the 1st International Conference on IoT in Urban Space; 2014:50-55; Rome, Italy.
55. Sotiriadis S, Bessis N, Asimakopoulou E, Mustafee N. Towards simulating the Internet of Things. Proceedings of the 2014 28th International Conference on Advanced Information Networking and Applications Workshops; 2014:444-448; Vancouver, BC, Canada.
56. Zeng X, Garg SK, Strazdins P, Jayaraman PP, Georgakopoulos D, Ranjan R. IOTSim: a simulator for analysing IoT applications. *Int J High Perform Syst Archit*. 2017;72:93-107.
57. Qayyum T, Malik AW, Khan Khattak MA, Khalid O, Khan SU. FogNetSim++: a toolkit for modelling and simulation of distributed fog environment. *IEEE Access*. 2018;6:63570-63583.
58. Fernández-Cerero D, Fernández-Montes A, Javier Ortega F, Jakóbik A, Widlak A. Sphere: simulator of edge infrastructures for the optimization of performance and resource energy consumption. *Simul Modelling Pract Theory*. 2020;101(1019663):101966.
59. Sonmez C, Ozgovde A, Ersoy C. Edgecloudsim: an environment for performance evaluation of edge computing systems. *Trans Emerg Telecommun Technol*. 2018;29(11):e3493.
60. Giang NK, Blackstock M, Lea R, Leung VCM. Developing IoT applications in the fog: a distributed dataflow approach. Proceedings of the 2015 5th International Conference on the Internet of Things (IOT); 2015:155-162; Seoul, South Korea.
61. Mayer R, Graser L, Gupta H, Saurez E, Ramachandran U. EmuFog: extensible and scalable emulation of large-scale fog computing infrastructures. Proceedings of the 2017 IEEE Fog World Congress (FWC); 2017:1-6; Santa Clara, CA.
62. Coutinho A, Greve F, Prazeres C, Cardoso J. Fogbed: a rapid-prototyping emulation environment for fog computing. Proceedings of the 2018 IEEE International Conference on Communications (ICC); 2018:1-7; Kansas City, MO.
63. Eisele S, Pettet G, Dubey A, Karsai G. Towards an architecture for evaluating and analyzing decentralized Fog applications. Proceedings of the 2017 IEEE Fog World Congress (FWC); 2017:1-6; Santa Clara, CA.
64. Banzai T, Koizumi H, Kanbayashi R, Imada T, Hanawa T, Sato M. D-Cloud: design of a software testing environment for reliable distributed systems using cloud computing technology. Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing; 2010:631-636; Melbourne, VIC, Australia.
65. De Oliveira RL, Schweitzer CM, Shinoda AA, Prete LR Using mininet for emulation and prototyping software-defined networks. Proceedings of the 2014 IEEE Colombian Conference on Communications and Computing (COLCOM); 2014:1-6; Bogota, Colombia.

66. Balasubramanian D, Dubey A, Otte WR, Emfinger W, Kumar PS, Karsai G. A rapid testing framework for a mobile cloud. Proceedings of the 2014 25nd IEEE International Symposium on Rapid System Prototyping; 2014:128-134; New Delhi, India.
67. Luckow A, Rattan K, Jha S. Exploring task placement for edge-to-cloud applications using emulation. Proceedings of the 2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC); 2021:79-83; Online, Australia.
68. Roy N, Dubey A, Gokhale A, Dowdy L. A capacity planning process for perform. assurance of component-based distributed systems. Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering; 2011:259-270; Karlsruhe, Germany.
69. De Maio V, Brandic I. Multi-objective mobile edge provisioning in small cell clouds. Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering; 2019:127-138; Mumbai, India.
70. Brogi A, Forti S. QoS-aware deployment of IoT applications through the fog. *IEEE Internet Things J*. 2017;4(5):1185-1192.
71. Cardellini V, Grassi V, Lo Presti F, Nardelli M. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Perform Eval Rev*. 2017;44(4):11-22.
72. Oh K, Chandra A, Weissman J. A network cost-aware geo-distributed data analytics system. Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID); 2020:649-658; Melbourne, Australia.
73. Goudarzi M, Wu H, Palaniswami M, Buyya R. An application placement technique for concurrent IoT applications in edge and fog computing environments. *IEEE Trans Mob Comput*. 2021;20(4):1298-1311.
74. Badri H, Bahreini T, Grosu D, Yang K. Energy-aware application placement in mobile edge computing: a stochastic optimization approach. *IEEE Trans Parallel Distrib Syst*. 2020;31(4):909-922.
75. He Z, Li K, Li K, Zhou W. Server configuration optimization in mobile edge computing: a cost-performance tradeoff perspective. *Softw Pract Exper*. 2021;51(9):1868–1895.
76. Khare S, Sun H, Gascon-Samson J, et al. Linearize, predict and place: minimizing the makespan for edge-based stream process. of directed acyclic graphs. Proceedings of the 4th ACM/IEEE Symposium on Edge Computing; 2019:1-14; Arlington, VA.
77. Xu X, Li D, Dai Z, Li S, Chen X. A heuristic offloading method for deep learn. edge services in 5G networks. *IEEE Access*. 2019;7:67734-67744.
78. Zhao P, Dan G. Joint resource dimensioning and placement for dependable virtualized services in mobile edge clouds. *IEEE Trans Mob Comput*. 2021.
79. Janßen G, Verbitskiy I, Renner T, Thamsen L. Scheduling stream processing tasks on geo-distributed heterogeneous resources. Proceedings of the 2018 IEEE International Conference on Big Data (Big Data); 2018:5159-5164; Seattle, WA.
80. Basic F, Aral A, Brandic I. Fuzzy handoff control in edge offloading. Proceedings of the 2019 IEEE International Conference on Fog Computing (ICFC); 2019:87-96; Prague, Czech Republic.
81. Tong L, Li Y, Gao W. A hierarchical edge cloud architecture for mobile computing. Proceedings of the 35th Annual IEEE International Conference on Computer Communications; 2016:1-9; San Francisco, CA.
82. Ma Y, Liang W, Li J, Jia X, Guo S. Mobility-aware and delay-sensitive service provisioning in mobile edge-cloud networks. *IEEE Trans Mob Comput*. 2020.
83. Deng S, Xiang Z, Taheriand J, et al. Optimal application deployment in resource constrained distributed edges. *IEEE Trans Mob Comput*. 2021;20(5):1907-1923.
84. Bagaa M, Taleb T, Bernabe JB, Skarmeta A. QoS and resource-aware security orchestration and life cycle management. *IEEE Trans Mob Comput*. 2020.
85. Nemeth B, Molner N, Martinperez J, Bernardos CJ, Oliva DA, Sonkoly B. Delay and reliability-constrained VNF placement on mobile and volatile 5G infrastructure. *IEEE Trans Mob Comput*. 2021.
86. Moubayed A, Shami A, Heidari P, Larabi A, Brunner R. Edge-enabled V2X service placement for intelligent transportation systems. *IEEE Trans Mob Comput*. 2021;20(4):1380-1392.
87. Zhao Y, Liu X, Tu L, Tian C, Qiao C. Dynamic service entity placement for latency sensitive applications in transportation systems. *IEEE Trans Mob Comput*. 2021;20(2):460-472.
88. Paul Martin J, Kandasamy A, Chandrasekaran K. CREW: cost and reliability aware eagle-whale optimiser for service placement in fog. *Softw Pract Exper*. 2021;50(12):2337–2360.
89. Naas MI, Parvedy PR, Boukhobza J, Lemarchand L. iFogStor: an IoT data placement strategy for fog infrastructure. Proceedings of the 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC); 2017:97-104; Madrid, Spain.
90. Wöbker C, Seitz A, Mueller H, Bruegge B. Fogernetes: deployment and management of fog computing applications. Proceedings of the NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium; 2018:1-7; Taipei, Taiwan.
91. Jindal A, Gerndt M, Chadha M, Podolskiy V, Chen P. Function delivery network: extending serverless computing for heterogeneous platforms. *Softw Pract Exper*. 2021;51(9):1936–1963.
92. Li Y, Dai W, Gan X, et al. Cooperative service placement and scheduling in edge clouds: a deadline-driven approach. *IEEE Trans Mob Comput*. 2021.
93. Gao B, Zhou Z, Liu F, Xu F, Li B. An online framework for joint network selection and service placement in mobile edge computing. *IEEE Trans Mob Comput*. 2021.
94. Saurez E, Hong K, Lillethun D, Ramachandran U, Ottenwälder B. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems; 2016:258-269; Irvine, CA.

95. Becker S, Schmidt F, Thamsen L, Ferrer AJ, Kao O. LOS: local-optimistic scheduling of periodic model training for anomaly detection on sensor data streams in meshed edge networks. Proceedings of the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS); 2021; Washington DC.
96. Fawcett L, Broadbent M, Race N. Combinatorial auction-based resource allocation in the fog. Proceedings of the 2016 5th European Workshop on Software-Defined Networks (EWSDN); 2016:62-67; Den Haag, Netherlands.
97. Mahmud R, Srirama SN, Ramamohanarao K, Buyya R. Profit-aware application placement for integrated fog–cloud computing environments. *J Parallel Distrib Comput*. 2020;135:177-190.
98. Zhou B, Srirama SN, Buyya R. An auction-based incentive mechanism for heterogeneous mobile clouds. *J Syst Softw*. 2019;152:151-164.
99. Li Q, Yao H, Mai T, Jiang C, Zhang Y. Reinforcement-learning- and belief-learning-based double auction mechanism for edge computing resource allocation. *IEEE Internet Things J*. 2020;7(7):5976-5985.
100. Gao G, Xiao M, Wu J, Huang H, Wang S, Chen G. Auction-based VM allocation for deadline-sensitive tasks in distributed edge cloud. *IEEE Trans Serv Comput*. 2019.
101. Ouyang X, Irwin D, Shenoy P. SpotLight: an information service for the cloud. Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS); 2016:425-436; Nara, Japan.
102. Shakarami A, Shahidinejad A, Ghobaei-Arani M. A review on the computation offloading approaches in mobile edge computing: a game-theoretic perspective. *Softw Pract Exper*. 2020;50(9):1719-1759.
103. Abdulsalam Y, Hossain MS. COVID-19 networking demand: an auction-based mechanism for automated selection of edge computing services. *IEEE Trans Netw Sci Eng*. 2020.
104. Chen L, Wu J, Zhang X, Zhou G. TARCO: two-stage auction for D2D relay aided computation resource allocation in HetNet. *IEEE Trans Serv Comput*. 2021;14(1):286-299.
105. Guo S, Dai Y, Guo S, Qiu X, Qi F. Blockchain meets edge computing: stackelberg game and double auction based task offloading for mobile blockchain. *IEEE Trans Veh Technol*. 2020;69(5):5549-5561.
106. Liwang M, Dai S, Gao Z, Tang Y, Dai H. A truthful reverse-auction mechanism for computation offloading in cloud-enabled vehicular network. *IEEE Internet Things J*. 2019;6(3):4214-4227.
107. Habiba U, Hossain E. Auction mechanisms for virtualization in 5G cellular networks: basics, trends, and open challenges. *IEEE Commun Surv Tutor*. 2018;20(3):2264-2293.
108. Sun W, Liu J, Yue Y, Zhang H. Double auction-based resource allocation for mobile edge computing in industrial Internet of Things. *IEEE Trans Ind Inform*. 2018;14(10):4692-4701.
109. Lee Y, Jeong S, Masood A, Park L, Dao NN, Cho S. Trustful resource management for service allocation in fog-enabled intelligent transportation systems. *IEEE Access*. 2020;8:147313-147322.
110. Kiani A, Ansari N. Toward hierarchical mobile edge computing: an auction-based profit maximization approach. *IEEE Internet Things J*. 2017;4(6):2082-2091.
111. Tasiopoulos AG, Ascigil O, Psaras I, Pavlou G. Edge-MAP: auction markets for edge resource provisioning. Proceedings of the 2018 IEEE 19th International Symposium on AWorldofWireless; 2018:14-22; Chania, Greece.
112. Zhang Y, Wang CY, Wei HY. Parked vehicle assisted VFC system with smart parking: an auction approach. Proceedings of the 2018 IEEE Global Communications Conference (GLOBECOM); 2018:1-7; Abu Dhabi, United Arab Emirates.
113. You M, Zhou H, Zhuang Y. Research on application of auction algorithm in internet of vehicles task scheduling under fog environment. Proceedings of the 2020 the 4th International Conference on Innovation in Artificial Intelligence; 2020:242-249; Xiamen, China.
114. Safianowska MB, Gdowski R, Huang C. Combinatorial recurrent multi-unit auctions for fog services. Proceedings of the 2016 International Computer Symposium (ICS).; 2016:736-741; Chiayi, Taiwan.
115. Peng X, Ota K, Dong M. Multiattribute-based double auction toward resource allocation in vehicular fog computing. *IEEE Internet Things J*. 2020;7(4):3094-3103.
116. Zu Y, Shen F, Yan F, et al. An auction-based mechanism for task offloading in fog networks. Proceedings of the 2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC); 2019:1-6; Istanbul, Turkey.
117. Baranwal G, Kumar D. DAFNA: decentralized auction based fog node allocation in 5G era. Proceedings of the 2020 IEEE 15th International Conference on Industrial and Information Systems (ICIIS); 2020:575-580; Rupnagar, India.
118. Jiao Y, Wang P, Niyato D, Suankaewmanee K. Auction mechanisms in cloud/fog computing resource allocation for public blockchain networks. *IEEE Trans Parallel Distrib Syst*. 2019;30(9):1975-1989.
119. Luong NC, Jiao Y, Wang P, Niyato D, Kim DI, Han Z. A machine-learning-based auction for resource trading in fog computing. *IEEE Commun Mag*. 2020;58(3):82-88.
120. Guo Y, Saito T, Oma R, Nakamura S, Enokido T, Takizawa M. Distributed approach to fog computing with auction method. Proceedings of the 34th International Conference on Advanced Information Networking and Applications (AINA-2020); 2020:268-275; Caserta, Italy.
121. Gharaibeh A, Khreishah A, Mohammadi M, Al-Fuqaha A, Khalil I, Rayes A. Online auction of cloud resources in support of the Internet of Things. *IEEE Internet Things J*. 2017;4(5):1583-1596.
122. Wang B, Ali-Eldin A, Shenoy P. LaSS: running latency sensitive serverless computations at the edge. Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing; 2020:239-251; Online, Sweden.
123. Rausch T, Hummer W, Muthusamy V, Rashed A, Dustdar S. Towards a serverless platform for edge AI. Proceedings of the HotEdge '19; 2019; Renton, WA.
124. Nastic S, Rausch T, Scekic O, et al. A serverless real-time data analytics platform for edge computing. *IEEE Internet Comput*. 2017;21(4):64-71.

125. Nastic S, Dustdar S. *Towards Deviceless Edge Computing: Challenges, Design Aspects, and Models for Serverless Paradigm at the Edge*. Springer; 2018.
126. Rausch T, Rashed A, Dustdar S. Optimized container scheduling for data-intensive serverless edge computing. *Future Generat Comput Syst*. 2021;114:259-271.
127. Machen A, Wang S, Leung KK, Ko BJ, Salonidis T. Live service migration in mobile edge clouds. *IEEE Wirel Commun*. 2017;25(1):140-147.
128. Wang S, Urgaonkar R, Zafer M, He T, Chan K, Leung KK. Dynamic service migration in mobile edge computing based on markov decision process. *IEEE/ACM Trans Netw*. 2019;27(3):1272-1288.
129. Aske A, Zhao X. Supporting multi-provider serverless computing on the edge. Proceedings of the 47th International Conference on Parallel Processing Companion; 2018:1-6; Eugene, OR.
130. Cheng B, Fuerst J, Solmaz G, Sanada T. Fog function: serverless fog computing for data intensive IoT services. Proceedings of the 2019 IEEE International Conference on Services Computing (SCC); 2019:28-35; Melbourne, VIC, Australia.
131. Pelle I, Paolucci F, Sonkoly B, Cugini F. Telemetry-driven optical 5G serverless architecture for latency-sensitive edge computing. Proceedings of the 2020 Optical Fiber Communications Conference and Exhibition (OFC); 2020:1-3; San Diego, CA.
132. Palade A, Mukhopadhyay A, Kazmi A, et al. A swarm-based approach for function placement in federated edges. Proceedings of the 2020 IEEE International Conference on Services Computing (SCC); 2020:48-50; Beijing, China.
133. Persson P, Angelsmark O. Kappa: serverless IoT deployment. Proceedings of the 2nd International Workshop on Serverless Computing; 2017:16-21; Las Vegas, NV.
134. Persson P, Angelsmark O. Calvin–merging cloud and IoT. *Proc Comput Sci*. 2015;52:210-217.
135. Baresi L, Mendonça DF, Garriga M. Empowering low-latency applications through a serverless edge computing architecture. Proceedings of the European Conference on Service-Oriented and Cloud Computing; 2017:196-210; Oslo, Norway.
136. Das A, Imai S, Patterson S, Wittie MP. Performance optimization for edge-cloud serverless platforms via dynamic task placement. Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID); 2020:41-50; Melbourne, VIC, Australia.
137. Cicconetti C, Conti M, Passarella A. Uncoordinated access to serverless computing in MEC systems for IoT. *Comput Netw*. 2020;172:107184.
138. Cicconetti C, Conti M, Passarella A. A decentralized framework for serverless edge computing in the Internet of Things. *IEEE Trans Netw Serv Manag*. 2020;18(2):2166-2180.

How to cite this article: Bermbach D, Bader J, Hasenburg J, Pfandzelter T, Thamsen L. AuctionWhisk: Using an auction-inspired approach for function placement in serverless fog platforms. *Softw Pract Exper*. 2022;52(5):1143-1169. doi: 10.1002/spe.3058